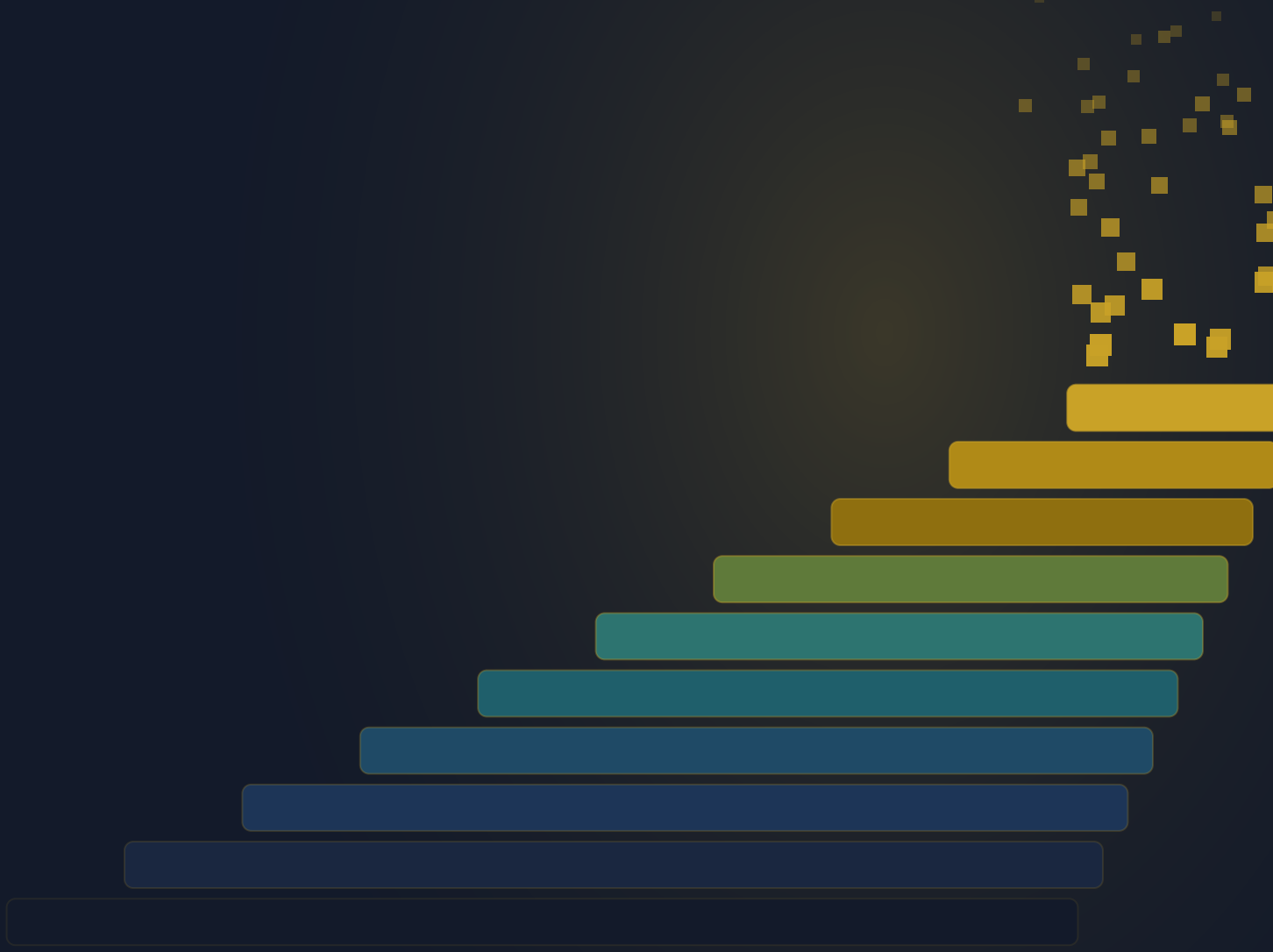
EVERY PIXEL MATTERS

BOOK 2



FROM CODE TO GLASS

From Code to Glass

Who Draws Your Screen, and What It Costs

Every screen your people trust is drawn by a chain of engines, processes and compositors that no single team owns. This book follows one frame from code to glass on Windows, macOS, Linux and Android, tells how each hop got this way and who pays for it, and gives a CTO a plain way to decide which hops to own, rent, pin or isolate, judged by security, cost, visibility and business value.

Juan Cardena

Enterprise AI & Agentic Systems Architect · blog.jcardena.com

CONTENTS

BEFORE YOU START

Reader's Brief: Security, Cost, Visibility, Value

PROLOGUE

One Tap, One Frame

PART I · THE FRAME, HOP BY HOP

01 Sixteen Milliseconds: The Frame Budget and the Ledger

02 From Markup to Geometry: Parse, Style and Layout

03 Text: Shaping, Fonts and the Glyph

04 Paint and Raster: From Display List to Pixels

05 Two Compositors, One Vsync

06 The Glass: Display, Color, HDR and Timing

PART II · BOUNDARIES AND TRUST

07 Processes, Sandboxes and the GPU as the New Kernel

08 Input and Hit-Testing: Who Gets the Click

09 Capture, Remote Display and the Trusted Pixel

10 When the User Is an Agent

PART III · RUNTIMES AND NATIVE STACKS

11 The Browser Inside the App: Eight Runtimes on One Ledger

12 The Native Path: Composition, Skia and the Rust Renderers

PART IV · MONEY, SHADOWS AND DEBT

13 Who Pays for the Engine: The Money Story

14 The Dark Pages: Antitrust, Exploits and Tracking

15 Cost and Debt: The Patch Bill, the Memory Bill, the Legacy Bill

PART V · AHEAD

16 The Screen Ahead: The OS, AI and the Renderer

CLOSING

Epilogue: From Reading the Frame to Owning It

APPENDICES

A Glossary
BEFORE YOU START
B The Frame Ledger, Blank
C The Five-OS Reference Table
D Measurements from the Source
E Reading List and Sources

Reader's Brief: Security, Cost, Visibility, Value

This brief tells you what the book will do for you and where to find it. It names the reader, the three companies that come back in every chapter, and the one worksheet the book builds hop by hop: the Frame Ledger. You will see one Ledger filled in for a real kind of screen before the first chapter starts. Read the brief in ten minutes, then follow the path for your role.

Who this book is for

It is written for the **CTO, CIO, CISO or chief architect at a company of 50 to 1,000 engineers**. Companies of that size ship screens that matter: a payment approval, a consent form, a check-in kiosk, a trading view. They rent almost every piece of software that draws those screens, and they are big enough that a wrong bet on one piece reaches thousands of devices at once.

You do not need to write graphics code. Every term is explained the first time it appears. You do need to answer for four things about the screens your company ships: their **security**, the **cost** of drawing and patching them, how much of their behavior you can actually **see**, and the **value** they create. Book 1 of this series, *From Bit to Pixel*, covered the whole stack from the processor to the screen. This book stays on one layer, the drawing of the screen itself, and goes deeper. You can read it without Book 1.

Two readers will get less from it. A team that builds browser engines already knows the mechanics, though it may find the cost and ownership chapters useful. A small startup with one web app should read the prologue, Chapter One and Chapter Eleven, and come back when its screens carry money or regulated consent.

Three companies, three screens

Advice about "the rendering pipeline" is easy to agree with and hard to use. So every chapter tests its ideas on the same three companies. All three are invented. Each is a composite of a common kind of business, and each chapter takes one of them as its lead.

> **LARKFIELD SOFTWARE, THE DESKTOP SOFTWARE MAKER**
>
> Larkfield sells a collaboration and analytics app that runs on about 400,000 customer desktops, on Windows, macOS and Linux, with a lighter companion app on iOS and Android. The desktop app is built on a bundled, Electron-style engine: Larkfield ships its own copy of a browser engine inside every install. Its questions are about that copy. Who patches it when a flaw is found, how much memory it costs every seat, and whether its hottest views should leave the engine for native code.

HALBROOK BANK, THE REGULATED COMPANY

Halbrook is a regional bank with 2,500 branch PCs. Its teller and advisor desktop app embeds Microsoft's Edge WebView2 runtime, which lets a Windows app show web pages drawn by the same engine as the Edge browser; the book calls it WebView2 after this first mention, and a chapter that needs to reintroduce it does so once as "Microsoft's Edge WebView2 runtime (WebView2)". It also has a mobile app on iOS and Android, and its advisors work on virtual desktops (VDI) from home. Its questions are about trust. What exactly did the teller see before tapping Approve, who else can read that screen, and can the bank show an auditor the answer on one page.

CORVANE HOTELS, THE DEVICE FLEET OPERATOR

Corvane runs 300 properties with about 3,000 self check-in kiosks on Windows IoT, Android tablets at the front desk, and signage screens on Linux. The machines live for years in lobbies with no engineer nearby. Its questions are about time. When does the engine pinned on each kiosk stop getting fixes, which GPUs and panels does the fleet really run, and what does a missed frame or a blank screen cost in site visits.

The value map

Here are twelve questions the book answers, one per company per theme, with the chapter that answers each. Find your company's row and start there.

Company	Security	Cost	Visibility	Value
Larkfield	How many engine copies do we ship, and how many days does each take to patch? **Chapter Eleven**	What does one engine per app cost our customers in memory and disk? **Chapter Fifteen**	What does our monitoring miss once a frame leaves the main thread? **Chapter One**	Which views are worth moving to native code? **Chapter Twelve**
Halbrook	Who can copy the payment screen, and what do the exclusion flags really stop? **Chapter Nine**	What does remote display cost us for advisors on VDI? **Chapter Nine**	Can we tell an agent's click from a teller's click? **Chapter Eight**	Can the amount the teller sees differ from the amount we store? **Chapter Three**
Corvane	When does the engine pinned on our kiosks stop getting fixes? **Chapter Eleven**	Do our kiosks draw on the GPU at all, or silently on the CPU? **Chapter Four**	Does anyone know what the panel actually showed? **Chapter Six**	When should our Linux signage move from X11 to Wayland? **Chapter Five**

What you will be able to do

- **Name every owner** that touches one frame of your most critical screen, hop by hop, on Windows, macOS, iOS, Linux and Android.
- **Trace that frame** with the platform's own tools, and write down which hops your monitoring can see and which it cannot.
- **Fill in a Frame Ledger** for the screen: one row per hop, with owner, payer, patch channel, sandbox and readers of the pixels.
- **Make an Own, Rent, Pin or Isolate call** for each hop, and defend it with evidence rather than with a framework preference.
- **Write the patch channel into your contracts**, so that the next engine flaw has a known owner and a known number of days.

A patch-channel clause can be short. Here is a sample, to adapt with your counsel: "The Supplier shall identify every third-party rendering component in the Product, including any bundled or embedded browser engine, with its version and update channel. When the upstream project releases a fix for a vulnerability rated High or Critical in such a component, the Supplier shall make a Product update containing the fix available within 14 calendar days, or within 7 calendar days if the vulnerability is listed in CISA's Known Exploited Vulnerabilities catalog, and shall notify the Customer of the fixed version and of any restart

needed to activate it." The day counts are examples to negotiate, not a standard; the parts that matter are the named component, the clock that starts at the upstream fix, and the activation step.

The Frame Ledger

Book 1 gave you the **Pixel Bill of Materials**: one row per component on the box, with who writes it and who can update it. This book adds its sibling, the **Frame Ledger**. The bill answers "what is on the box, and who owns it?" The Ledger answers "how does this one screen get drawn, who touches the frame on the way, and who can read the result?" Each Ledger row points to the bill's row for the component it uses, so the two documents stay linked and never repeat each other.

A Ledger covers one critical screen. Its rows are the hops one frame takes: **0 Source** (markup, code or native scene), **1 Parse**, **2 Style**, **3 Layout**, **4 Text**, **5 Paint**, **6 Raster**, **7 Engine compositor**, **8 OS compositor**, **9 Scanout and display** and **10 Glass and readers**, the physical screen and whoever looks at it or copies it. When one hop needs more than one row, the rows are numbered as sub-rows, such as 4a and 4b. Two side rows follow the frame's other directions: **Input**, the path from a finger back to your code, and **Capture**, every path that copies the frame out. Not every screen uses every hop. A native app may have no parse step, and a full-screen app can sometimes skip the OS compositor. The numbers give an order, not a strict sequence: hops 3 and 4 run together, because layout asks the text stack to shape runs of text in order to find the line breaks. Hop 4 keeps its own rows anyway, because its owners and patch channels differ from layout's. The Ledger records the path your screen really takes.

The core worksheet has six columns, small enough for one page.

Column	What it asks
Hop and component	The real component, by name and version policy: evergreen (updates itself), pinned (a version you chose), OS-tied (updates with the OS) or bundled (shipped inside your app).
Owner and who pays	Who writes the code, and who funds that work: a vendor, a foundation, or you.
Who patches, and how fast	The update channel, and the measured days from an upstream fix to a fix running on your devices.
Process and sandbox	Which process runs the hop, and how tightly the OS limits what that process can do.
Who can read it	Who can see or copy what this hop produces: the user, capture tools, screen-reading agents, assistive technology, a remote-display client.
Decision	A tag in capitals, OWN, RENT, PIN or ISOLATE (primary tag first when there are two), then a one-line reason.

The four decisions mean this. To **own** a hop is to write or carry the code yourself, such as your own page or your own font file. To **rent** it is to use the vendor's component as shipped, on the vendor's schedule. To **pin** it is to choose a fixed version and update it on your own schedule, which buys stability and costs you the patch work. To **isolate** it is to keep it in a separate process, sandbox or desktop so that its failure or its readers cannot reach the rest. Combinations are normal. Appendix B holds the blank worksheet and an extended sheet with six more columns: the bill-of-materials row, untrusted input, frame cost, observable by, tested recovery and the accountable person.

One Ledger, filled in

Here is the Ledger for Halbrook's branch payment-approval screen. The screen is a web page inside the teller app, drawn by WebView2 on Windows. Two terms in it need one sentence each. WebView2 ships in two modes: **Evergreen**, one shared copy, installed per machine or per user, that Microsoft updates automatically with Edge's updates, and **Fixed Version**, a specific copy of the runtime that the app maker downloads, packages with the app and updates on its own schedule. Halbrook uses Evergreen, installed machine-wide. An **integrity level** is a label Windows puts on every process; a process at a lower level cannot write to objects at a higher one, ordinary apps run at Medium, and Chromium places its most exposed processes below that, at Low or at the lowest level, Untrusted. The prologue follows one frame of this screen from the teller's tap to the glass.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision
0 to 5 Source to paint: the approval page, drawn by Blink in WebView2, Evergreen	Halbrook writes and pays for the page. Blink is written upstream in Chromium, a Google-led project; Microsoft builds and ships the runtime	The page: Halbrook's release train. The engine: Microsoft, with Edge's updates. An update is used only after the app restarts or releases its old environment	Renderer process; Chromium documents its renderers at Windows' lowest, Untrusted, integrity level	The page's own scripts and any third-party script it loads; assistive technology and screen agents, through the accessibility tree	**OWN, RENT:** own the page and allow no third-party script on this screen; rent the engine, with an app-restart rule
4a Text shaping: Blink shapes text with HarfBuzz	HarfBuzz and Blink upstream; Microsoft ships the runtime	Microsoft, with the engine's updates; active after the app restarts	Renderer process	Page scripts, assistive technology and screen agents read the shaped text	**RENT:** the shaping code comes with the runtime
4b Font supply: Windows DirectWrite supplies system fonts, metrics and glyph rasterization, reached through the browser process's font proxy; Halbrook's licensed brand font	Microsoft for DirectWrite; Chromium upstream for the proxy; the font foundry, paid by Halbrook's license	Microsoft, for DirectWrite through Windows Update and for the proxy with the engine; Halbrook, for the font file, only when it replaces it	The sandboxed renderer cannot open the system font collection, so it asks the browser process over IPC for each system and fallback font (Chapter Three)	Not readable as such; its output is the shaped text in row 4a	**OWN:** ship the licensed font file in the app; block remote fonts on this screen, because a font from outside is untrusted input to the parser
6 to 7 Raster and engine compositor: Chromium raster and Viz, Chromium's display compositor, which assembles the final frame from every layer in the GPU process; presented through DirectCompositi on	Chromium upstream; Microsoft ships; the GPU driver vendor for the driver underneath	Microsoft, for the runtime; the GPU driver vendor, for the driver, delivered through Windows Update or the PC maker	GPU process, documented at Low integrity, a looser sandbox than the renderer's because it must talk to the driver	Not readable by the page; the driver sees every command	**RENT, ISOLATE:** rent the runtime's raster path; keep the GPU sandbox on and keep sandbox-disabling flags out of the app's configuration
8 OS compositor: Desktop Window Manager (DWM)	Microsoft	Windows Update, monthly security releases	`dwm.exe`, always on since Windows 8, though independent flip and multiplane overlay scanout can take a frame past it	The teller, Windows capture APIs, recording and screen-agent tools	**RENT:** set display affinity on the window and record it as a deterrent, not a control

9 to 10 Scanout, display, glass and readers: branch monitor and customer-facing display	The display maker; Halbrook buys it	The display maker, through monitor firmware, rarely or never	None	Teller, customer, anyone with a camera	**OWN** the policy: an approved-panel list; amount and payee shown as plain text and checked against the transaction record
Capture side row: call recording and assistant tools on branch PCs	Several vendors, paid by Halbrook	Each vendor, on its own release channel	Each vendor's own process, often privileged	Those vendors and whoever reads their stores	**ISOLATE:** no screen-reading agents on the approval desktop

Read the table across and a risk appears that no single row shows. The pixels the teller trusts pass through at least five owners, and the row marked OWN for the page needs care: Halbrook writes the page's script, which runs at hop 0, and its markup and CSS, which are input that controls what hops 1 to 5 do, but the code that executes those hops is Blink's. The fastest owner can fix a flaw within days while the slowest, the monitor, may never be updated at all. The engine update is not even active until the teller app restarts: Microsoft's documentation says a continuously running app "will continue to use the previous version". None of this is a reason to drop WebView2. It is a reason to add two lines to the bank's operating rules: a restart window for the teller app, and a list of tools allowed to read the approval screen.

Two details in the table are worth naming now, because they recur. Microsoft's documentation for display affinity, the Windows setting that hides a window from capture, says there is "no guarantee" that it "will strictly protect windowed content, for example where someone takes a photograph of the screen". And the European rule on payment authentication, Delegated Regulation (EU) 2018/389, Article 5, requires that "the payer is made aware of the amount of the payment transaction and of the payee". It asks what the payer was made aware of. It does not ask anyone to attest what a display showed. Chapter Nine takes that gap apart.

Reading paths by role

Role	Start with	Then	Keep for reference
CTO or VP Engineering	Prologue, Chapter One	Chapters Eleven, Thirteen and Fifteen	The Business implications box at the end of each chapter
Chief architect	Part I in order: Chapters One to Six	Chapters Seven, Eleven and Twelve	Appendix C, the five-OS reference table
CISO or security architect	Chapters Three, Seven and Nine	Chapters Eight, Ten and Fourteen	The Security lens box in each chapter
End-user computing or VDI owner	Chapters One, Four and Five	Chapters Nine and Fifteen	The Process and sandbox column of each Ledger
FinOps or procurement	Chapters Thirteen and Fifteen	Chapter Eleven	Appendix D, the measurements
AI platform lead	Chapter Ten	Chapters Eight, Nine and Sixteen	The Who can read it column of each Ledger

The one idea to carry

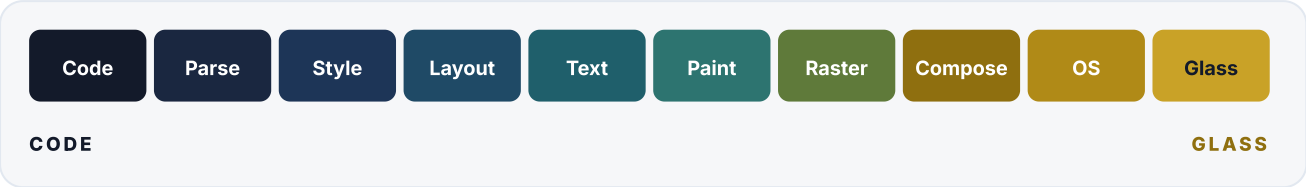
Every hop between your code and the glass has an owner. Owning more hops does not make a screen safer or faster by itself, and renting a hop is not a failure. What the Ledger shows is narrower and more useful.

> *The pixel your user trusts passed through many owners. Knowing who touches the frame, and how fast each one can fix it, is an architecture decision.*

Keep that sentence next to every decision in this book. When someone asks "should we build this screen in X?", ask instead: who will draw it, who can read it, and when one of those owners ships a flaw, how many days until the fix is running on our devices?

PROLOGUE

One Tap, One Frame



A teller at a Halbrook Bank branch checks a wire transfer on her screen: the payee, the amount, the account. She taps Approve. The button darkens, a green tick appears, and she moves on to the next customer. It feels like one event. It is a relay through the hops this book numbers 0 to 10, plus an input path on the way in and copies on the way out, and only one of them, hop 0, runs code the bank wrote. The bank's markup and CSS steer what the later hops do, but they are input to those hops, not the code that executes them.

This book follows that relay. It starts at the tap and ends at the glass, the physical screen and the eyes in front of it. Between the two sit an input system, a browser engine split across several processes, a compositor inside the engine, a compositor inside the operating system, a graphics driver and a display. Each hop has an owner, a payer, a patch channel and a list of who can read what it produces. When a hop changes, the bank's screen changes with it, whether or not the bank was ready.

The teller's app is a Windows desktop program that embeds Microsoft's Edge WebView2 runtime (WebView2) in its Evergreen mode, so the runtime updates itself. The approval screen is a web page inside that app. Here is what happens to one frame of it. The section headings use the hop numbers of the Reader's Brief: 0 Source, 1 Parse, 2 Style, 3 Layout, 4 Text, 5 Paint, 6 Raster, 7 Engine compositor, 8 OS compositor, 9 Scanout and display, 10 Glass and readers, plus the Input and Capture side rows.

Input side row: the tap becomes a message

The touch screen or mouse reports a contact to Windows. Windows turns it into a **pointer message**, such as `WM_POINTERDOWN`, part of the pointer input model Microsoft introduced with Windows 8. The message lands in the queue of the thread that owns the window under the finger. Microsoft's documentation states the rule plainly: a window belongs to the thread that created it, and each thread that creates a window runs its own **message loop** to take messages off the queue and hand them out.

Owner: Microsoft, for the input stack, and the device maker for the touch controller's firmware. **Who patches:** Windows Update for the first; rarely anyone for the second.

Input side row, continued: the runtime takes the event

In WebView2's default **windowed hosting** mode, the web content sits in its own window inside the teller app. Microsoft's documentation says that in this mode "the OS sends the input to the WebView2". The event goes to the runtime's **browser process**, `msedgewebview2.exe`, which decides which part of the page is under the finger and forwards the event over an internal message channel to the process that runs the page.

The teller app did not draw any of this. It owns the window frame around the page, and it chose the runtime. Microsoft's process-model documentation describes what sits underneath: each user data folder gets "a single browser process", plus helper processes such as the GPU process, and every WebView2 control created from the same environment and folder shares them. Renderer processes are created per site, and per frame where Site Isolation applies, and one renderer "can serve frames in multiple" controls. So the bank's single app is, underneath, a small family of processes, and the family grows with each separate user data folder and each distinct site.

Owner: the Chromium project, a Google-led open-source project, writes most of the engine; Microsoft builds, signs and ships the runtime. **Who patches:** Microsoft, with the same updates that Microsoft Edge receives.

Hops 0 to 5: the page decides what changed

The page runs in a **renderer process**. Chromium's sandbox design documents that its renderers run at Windows' lowest integrity level, called Untrusted, so a flaw in them has little to reach. On the renderer's main thread, the engine, called **Blink**, does four things in order.

- **Hit-test** (the end of the Input row): it walks its tree of boxes to find the element under the tap, the Approve button.
- **Run the handler** (hop 0, Source): it calls the bank's JavaScript, which records the approval and changes the button's state. This is the one hop that runs code Halbrook wrote. No new markup arrives, so hop 1, Parse, has no work in this frame.
- **Style and layout** (hops 2 and 3): it recomputes which CSS rules apply to the changed elements, then their size and position. Hops 3 and 4 run together: layout asks the text stack to shape runs of text to find where lines break.

- **Text and paint** (hops 4 and 5): it shapes the button's label with the HarfBuzz library, using fonts and glyph drawing supplied by Windows DirectWrite, then records a **display list**, a list of drawing commands, not yet pixels.

The result is handed to the renderer's **compositor thread**, which arranges the page into layers; this is where hop 7, the engine compositor, begins. Chromium calls this hand-off a **commit**. Its documentation explains that the main thread is blocked while the data is copied, so the compositor can keep scrolling and animating while the main thread works on the next change.

Owner: Chromium upstream for the engine; Halbrook for the page and its scripts; the font foundry for the bank's licensed brand font. **Who patches:** Microsoft for the engine, Halbrook for the page, and nobody for the font unless the bank replaces the file.

Hops 6 and 7: the GPU process makes pixels

The layers go to the runtime's **GPU process**. There, the display list is turned into pixels on the graphics processor, a step called **raster** (hop 6). Then **Viz**, Chromium's display compositor, assembles the frame from every layer and every process that contributed to it, completing hop 7. On Windows, Chromium presents that frame through **DirectComposition**, Microsoft's interface for handing composed surfaces to the system.

This is the hop with the widest gap between who writes the code and who can fix it. The GPU process must talk to the graphics driver, which the GPU maker writes. For that reason its sandbox is looser: Chromium's design notes place it at Low integrity, one step above the renderer's Untrusted. Calling the GPU process "the new kernel" is a metaphor, and an imperfect one. It is not the most privileged code on the machine. It is the code that sits closest to the driver while still taking orders from web content.

Owner: Chromium upstream and Microsoft for the GPU process; the GPU maker for the driver. **Who patches:** Microsoft for the runtime; Windows Update or the PC maker for the driver.

Hop 8: Windows composes the desktop

The teller does not see the page alone. She sees the page, the teller app around it, a notification, the taskbar and the cursor. The **Desktop Window Manager** (DWM) builds that image. Microsoft's documentation describes the model introduced with Windows Vista: windows "no longer draw directly to the screen", their drawing goes to off-screen surfaces, and DWM renders those into one desktop image. Since Windows 8, DWM can no longer be switched off; the old function to disable it is documented as having "no effect".

DWM is not always in the path for every pixel. Windows can let a suitable full-screen app flip its frames straight to the display, and modern display hardware can scan out some surfaces on their own planes, called overlays. A teller app in a normal window usually goes through DWM.

Owner: Microsoft. **Who patches:** Windows Update.

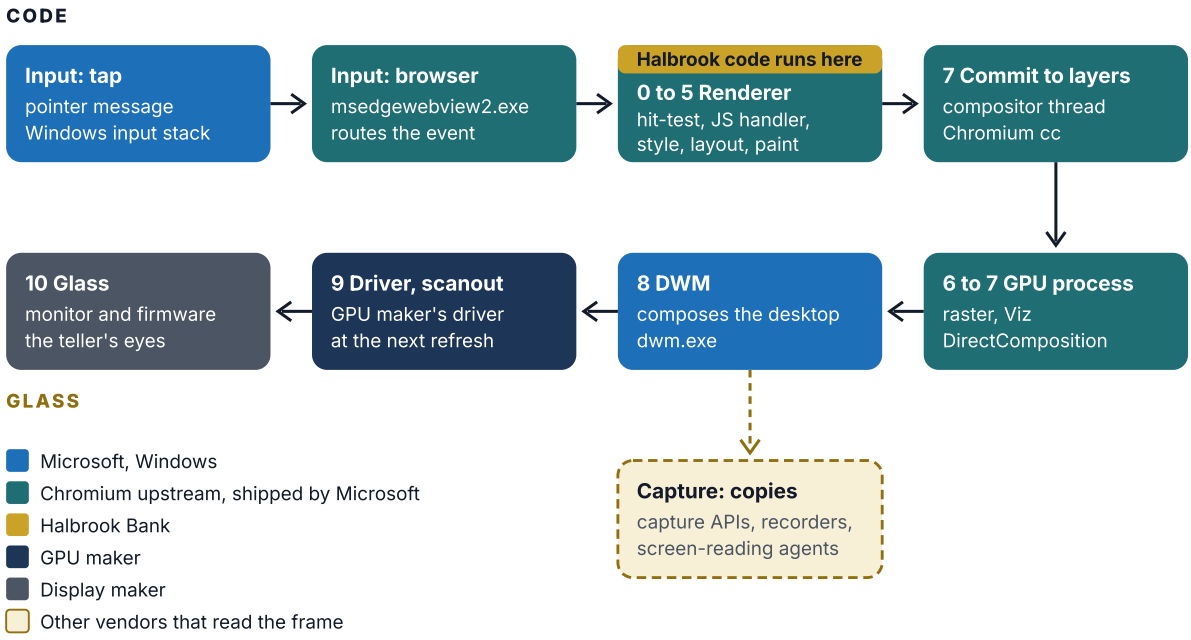
Hops 9 and 10: scanout, display and glass

DWM hands the finished desktop to the graphics driver, which tells the display hardware to show it at the next refresh. The display controller sends it down the cable, and the monitor's own electronics light the pixels. At 60 refreshes a second, a new image can appear every 16.7 milliseconds. Chapter One explains why the tick usually appears a few refreshes after the tap, not one.

Owner: the GPU maker for the driver; the display maker for the monitor and its firmware. **Who patches:** Windows Update or the PC maker for the driver. The monitor's firmware is rarely updated in the field, and nothing in the frame reports what the panel actually showed.

FIGURE P.1

One frame of the approval screen, and who owns each hop



The common composed path for a web page in WebView2's windowed hosting on Windows, numbered with the Brief's hops. Hop 7 starts on the renderer's compositor thread and finishes in Viz, inside the GPU process. Five owners touch the frame before the teller sees it, and more read copies of it after. Only the gold band is code the bank wrote.

Capture side row: the copies

The frame does not stop at the glass. Windows offers several ways to copy what is on screen, among them the `Windows.Graphics.Capture` and Desktop Duplication interfaces. A call-recording tool, a support tool or an AI assistant on the branch PC can use them. A screen-reading agent can also skip the pixels and read the page's structure through UI Automation, Windows' accessibility interface. Each of these is a reader of the approval screen that the bank must count.

An app can ask Windows to hide a window from capture, with a setting called display affinity. Microsoft is careful about what that setting promises. Its documentation says that "unlike a security feature or an implementation of Digital Rights Management (DRM), there is no guarantee" that it will "strictly protect windowed content, for example where someone takes a photograph of the screen". A copy of the screen is also not the same as what the panel showed: it is a capture of a composed surface, taken at one moment, and later color processing or overlays can make the two differ.

Who touched the frame

Hop	Owner	Who ships the fix
Input side row: pointer input	Microsoft; touch controller maker	Windows Update; firmware rarely
Input side row (browser process) and hops 2 to 7: renderer, compositor, GPU process (hop 1, Parse, did no work in this frame)	Chromium upstream (Google-led); Microsoft builds the runtime	Microsoft, evergreen; active after the app restarts
0: the approval page and its scripts	Halbrook	Halbrook's release train
4: brand font	Font foundry, under Halbrook's license	Halbrook, by replacing the file
8: DWM	Microsoft	Windows Update
9: graphics driver	GPU maker	Windows Update or the PC maker
9 to 10: monitor	Display maker	Rarely anyone
Capture side row: recorders and agents	Each tool's vendor	Each vendor's channel

The bank wrote one row. It chose several more: the runtime, its evergreen mode, the font, the monitors and the tools allowed on the desktop. The rest came with Windows and with the hardware. That split is not a weakness. It is the normal shape of a modern screen, and the reason this book exists.

> *The pixel your user trusts passed through many owners. Knowing who touches the frame, and how fast each one can fix it, is an architecture decision.*

The same frame elsewhere

On a Mac, the same page in Apple's WKWebView would be drawn by WebKit, composed by Core Animation and the WindowServer, and shown through Metal. On an iPhone, apps draw web content with WebKit unless Apple grants them an alternative-engine entitlement, which it offers only for users in the European Union (from iOS 17.4) and in Japan (from iOS 26.2). On a Linux desktop under Wayland, the compositor is the display server, and an app receives a frame callback when it is a good time to draw. On Android, the System WebView draws with Chromium and SurfaceFlinger composes the screen on each vertical sync. The hops have different names and the owners change. The shape, and the questions, stay the same.

The map of the book

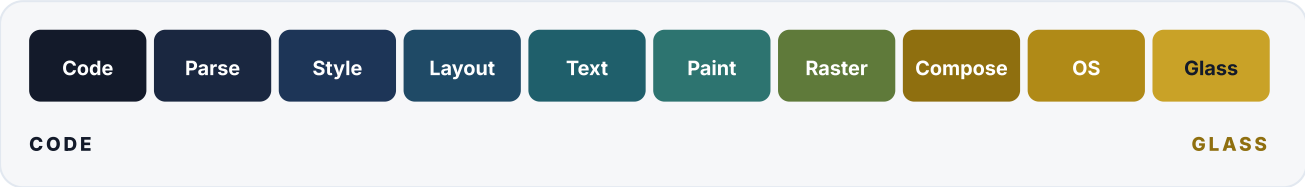
The book follows that shape. Each chapter adds rows to the Frame Ledger introduced in the Reader's Brief.

- **Part I, The Frame, Hop by Hop** (Chapters One to Six): the frame budget and the Ledger method; parse, style and layout; text; paint and raster; the two kinds of compositor and when there is only one; the glass, with color, HDR and timing.
- **Part II, Boundaries and Trust** (Chapters Seven to Ten): processes and sandboxes; input and hit-testing, or who gets the click; capture, remote display and the trusted pixel; and what a screen means when the user is an agent.
- **Part III, Runtimes and Native Stacks** (Chapters Eleven and Twelve): six embedded web runtimes on one Ledger, then the native path and the new Rust renderers.
- **Part IV, Money, Shadows and Debt** (Chapters Thirteen to Fifteen): who pays for the engines; the court records, exploits and tracking that brochures leave out; and the patch, memory and legacy bills.
- **Part V, Ahead** (Chapter Sixteen): where the screen is heading with AI, dated and labeled as opinion.

Chapter One starts with the clock that every hop must beat: the next refresh of the display.

CHAPTER ONE

Sixteen Milliseconds: The Frame Budget and the Ledger



When a customer says your screen is slow, which number do you ask for? Most teams ask for the time their own code took, because that is what their monitoring shows. This chapter argues for a different question: did the frame reach the glass by its deadline, and which owner made it late? It then gives you the worksheet to answer it for any critical screen, the Frame Ledger.

What a frame is, and who sets its deadline

A display does not show a moving picture. It shows a still image, then replaces it, many times a second. Each still image is a **frame**, and the number of replacements per second is the **refresh rate**, measured in hertz (Hz). The signal that marks each replacement is called **vertical sync**, or **vsync**. A frame that is ready before the next vsync is shown on time. A frame that misses it waits for the one after, and the user sees the old image twice. That repeat is a dropped frame, and users feel it as stutter, or **jank**.

The arithmetic is simple. At a fixed 60 Hz, one second divided by 60 gives 16.7 milliseconds per frame. At 120 Hz it is 8.3 milliseconds. Android's Profile GPU Rendering tool draws a horizontal green line that "represents 16.67 milliseconds", and its documentation says that "to achieve 60 frames per second, the vertical bar for each frame needs to stay below this line".

That number is a deadline only on a display that refreshes at a fixed rate. Many screens no longer do. Apple's developer guidance for ProMotion gives a range of 10 to 120 Hz on supported iPhones and 24 to 120 Hz on iPad Pro, and Pro iPhones with the Always-On display, starting with iPhone 14 Pro in September 2022, drop to a 1 Hz refresh rate for that mode. Apple tells developers that "your app can't presume a specific refresh rate at any time". Windows 11 has a Dynamic Refresh Rate feature that "automatically adjusts the refresh rate based on what you are doing at the time", on displays that support variable

refresh and at least 120 Hz. On such hardware the deadline moves. The useful rule is not "16.7 milliseconds" but "before the next vsync the system has promised", whatever that turns out to be.

Notice who sets the deadline. It is not your code. It is the display and the operating system's compositor, the program that assembles every window into one screen image. Your app is one runner in a relay that has to finish before a clock it does not own.

The loop inside one interval

Inside each interval, a typical frame passes through the same stages, whatever the platform calls them. The numbers are the Ledger's hop numbers from the Reader's Brief.

- **Input (side row):** pending touches, clicks and keys are handed to the app.
- **Script or handlers (hop 0, Source, and hop 1, Parse, when new markup arrives):** the app's code reacts and changes its state.
- **Style and layout (hops 2 and 3):** the engine or toolkit works out what each element looks like and where it goes.
- **Text (hop 4):** characters are shaped into positioned glyphs. Hops 3 and 4 run together rather than one after the other: layout asks the text stack to shape runs of text to find where each line breaks, then places the lines.
- **Paint (hop 5):** it records drawing commands, a display list.
- **Raster (hop 6):** the commands become pixels, usually on the graphics processor (GPU).
- **Composite (hops 7 and 8):** layers are assembled into a frame, first by the engine compositor if there is one (7), then by the OS compositor (8).
- **Present (hops 9 and 10):** the finished image is scanned out to the display at the next vsync and reaches the glass and whoever reads it.

The first stages, up to paint, usually run on one thread, the app's **UI thread** or main thread. That is not an accident. Mainstream UI frameworks tie their UI objects to one thread. Microsoft's Win32 documentation says applications "can include a message loop in each thread that creates a window", because a window belongs to the thread that made it. Android's documentation is blunter: "Bear in mind that the Android UI toolkit is not thread-safe. So, don't manipulate your UI from a worker thread." Rendering itself may happen on other threads and in other processes, but the work that decides what to draw is serialized on one.

A browser engine splits the work further. In Chromium, the engine inside Chrome, Edge, Electron and Android's System WebView, the renderer's main thread runs script, style, layout and paint, then **commits** its result to a separate compositor thread. Chromium's documentation describes a **scheduler** that decides when to produce frames and "maintains a deadline by which it expects its embedder to respond". The pace comes from below: each frame begins with a **BeginFrame** signal, which the documentation says is "driven by a viz::BeginFrameSource", itself driven by the display compositor in the GPU process. When the main thread is slow, the scheduler may draw without waiting for it, keeping scrolling smooth while the page catches up.

This is why "the app is slow" so often means a missed deadline somewhere else. A handler can finish quickly and the frame can still be late because raster took too long, because the GPU was busy with another app, or because the OS compositor missed its own deadline.

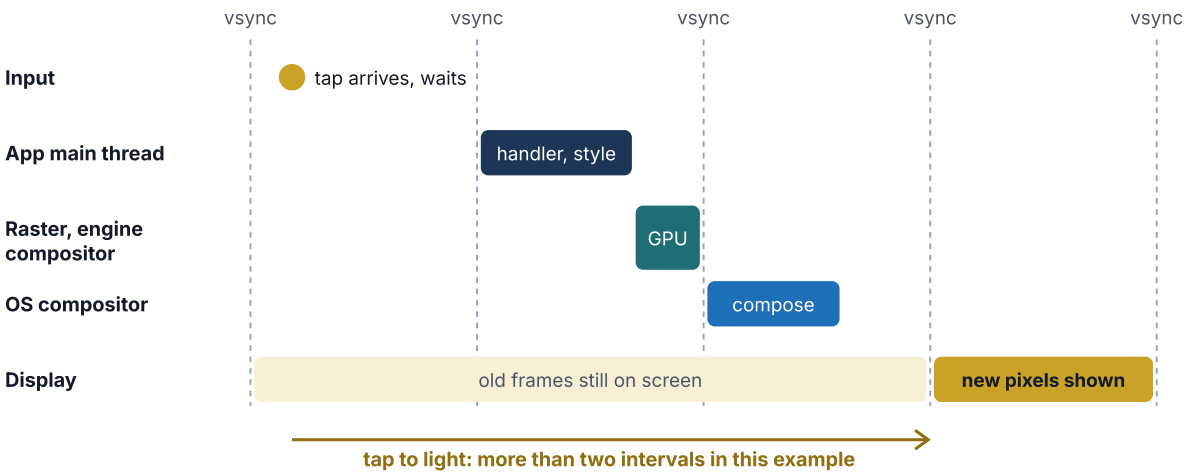
Input to photon: why a tap takes more than one frame

The time from a user's touch to the first changed pixel is called **input-to-photon latency**. It is longer than one frame budget, because the stages are pipelined across intervals. Android's platform documentation describes the steady state: on each vsync, "the display begins showing frame N while SurfaceFlinger begins compositing windows for frame N+1", and "the app handles pending input and generates frame N+2". Its conclusion: "The latency is at least two frames." Platforms shift the app's and compositor's signals relative to the hardware vsync to win some of that back, but the pipeline stays.

Queues add more. Microsoft's guidance for DirectX swap chains, the buffers an app draws into, warns that with the default queuing "the game is always waiting almost a full extra frame between the time it renders and the time it presents each frame". Its fix, the **waitable swap chain** added with Direct3D 11.2 and DXGI 1.3, lets the app wait until the system is ready and then draw from fresh input, with a default queue of one frame.

FIGURE 1.1

One tap, several refresh intervals



A simplified pipelined frame. Each stage starts on a vsync and hands its result to the next stage at the following one. Every stage can be on time and the user still waits several intervals. At 60 Hz one interval is 16.7 ms; at 120 Hz it is 8.3 ms.

How much latency do people notice? Less than most teams assume, but the answer depends on the task, so quote each study for its own task. Both studies below used touch screens in a laboratory, with a one-millisecond reference system; neither measured typing, reading or scrolling a form.

- **Dragging, perception.** Ng, Lepinski, Wigdor, Sanders and Dietz ("Designing for Low-Latency Direct-Touch Input", UIST 2012) built a touch prototype with a baseline latency of one millisecond and asked people to compare dragging at different latencies. The full paper is not openly available, so its results are given here as the same research group reported them in later papers: a perception floor of about six milliseconds on average, with individual thresholds from about 2 to 11 milliseconds. The group's CHI 2013 paper cites the lowest, "as little as 2.38ms of latency when dragging".
- **Tapping and landing.** Jota, Ng, Dietz and Wigdor ("How Fast is Fast Enough? A Study of the Effects of Latency in Direct-Touch Pointing Tasks", CHI 2013) found that dragging performance got worse as latency rose from 10 to 25 and 50 milliseconds. For the moment a finger lands on the screen, their participants' just-noticeable difference ranged from 20 to 100 milliseconds, with a mean of 64; no participant noticed less than 20.

The lesson for a CTO is not a single target number. It is that the frame budget is only one part of what users feel, and the rest lives in queues that no one team owns.

Tracing one frame on each OS

You cannot manage what your tools cannot see. Application performance monitoring usually times the code it wraps: a handler, a request, a render function. It rarely follows the frame into raster, the GPU, the OS compositor and the display. Each platform has a tool that does. Learn the one for your screens, because the Ledger's evidence comes from it.

Windows. Windows Performance Recorder (WPR) captures system-wide traces. Its built-in profiles include **GPU activity** and **Desktop composition activity**, which records "all desktop composition activity" by DWM. Windows Performance Analyzer (WPA) opens the trace and lines up CPU threads, GPU work and composition. For apps built on Microsoft's Edge WebView2 runtime (WebView2), Microsoft publishes a `WebView2.wprp` recording profile for the runtime's own events, and the Edge DevTools Performance tool, opened by right-clicking the control and choosing Inspect or through `edge://inspect`, shows the page's main-thread work. PresentMon, an open-source tool from Intel under the MIT license, reads the same Event Tracing for Windows data to report CPU, GPU and display frame times and latencies per app.

Linux, Chromium and Android. Perfetto is Google's open-source tracing suite, built to be "the default tracing system for the Android OS and the Chrome Browser", with OS-level probes on Linux as well. Its viewer at `ui.perfetto.dev` also opens Chrome's older JSON trace format, so a Chromium trace from Electron, CEF or Chrome can be read in the same tool. On Android 12 and later, Perfetto's **FrameTimeline** data source, a module inside SurfaceFlinger, labels each late frame with its cause, such as "AppDeadlineMissed" or a SurfaceFlinger deadline missed on the CPU or GPU.

Android, quickly. For a first look without a full trace, `adb shell dumpsys gfxinfo <package> framestats` prints nanosecond timestamps for each of the last 120 frames, a feature added in Android 6.0, and the on-device Profile GPU Rendering bars show each frame against the 16.67 millisecond line.

macOS and iOS. Xcode's Instruments has an **Animation Hitches** template. Apple defines a **hitch** as a frame that reaches the screen late and reports a **hitch time ratio** in milliseconds of hitch per second, which makes runs of different length comparable; in its 2020 developer session it treated under 5 milliseconds per second as good. In shipping apps, MetricKit reports the same measure from users' devices.

Frame loop	Windows	macOS	iOS	Linux	Android
UI thread loop	Win32 message loop; WinUI DispatcherQueue	Main run loop (AppKit)	Main run loop (UIKit)	GLib main loop (GTK); Qt event loop	Looper on the main thread
Frame pacing signal	DWM presentation; DXGI waitable swap chain	CADisplayLink (macOS 14 and later)	CADisplayLink (since iOS 3.1)	Wayland `wl_surface.frame` callback	Choreographer (since API level 16)
Refresh rate	Fixed, or Dynamic Refresh Rate on VRR displays of 120 Hz and up	Display dependent	ProMotion: 10 to 120 Hz on supported iPhones; 1 Hz in Always-On mode on Pro models from iPhone 14 Pro	Compositor and display dependent	Device and app dependent
Trace one frame	WPR and WPA; PresentMon; `WebView2.wprp`	Instruments	Instruments; MetricKit in the field	Perfetto	Perfetto; `gfxinfo framestats`
Who names the late frame	WPA composition views; PresentMon	Animation Hitches	Animation Hitches	Trace analysis by hand	FrameTimeline (Android 12 and later)

The table changes one decision: which tool your team must know per platform before it can sign off a critical screen. If a platform's row is blank in your team's skills, its Ledger rows will say "not observed", and that is worth knowing before a regulator or a customer asks.

HOW IT GOT THIS WAY

The refresh clock is older than any software on this path. A cathode-ray tube drew its picture line by line with an electron beam, then paused while the beam returned to the top, a pause called the vertical blanking interval. Changing the image during that pause avoided a torn picture, so graphics systems learned to wait for it. Flat panels have no beam, but the signal and the habit survived: every system in this book still paces frames to a vertical sync.

The single UI thread is nearly as old. The Win32 message loop that Microsoft still documents, built from `GetMessage`, `TranslateMessage` and `DispatchMessage`, descends from the Windows API that first shipped with Windows 1.0 on 20 November 1985. Android added its `Choreographer` in API level 16, Android 4.1, to coordinate "animations, input and drawing" on the timing pulses from the display. Apple's `CADisplayLink` arrived in iOS 3.1 and reached macOS only in version 14. The decision this history forces on you today: work that decides what to draw must reach the UI thread, so every library, script and agent you add to a screen competes for the same few milliseconds.

What hasn't changed: the BeginFrame-paced scheduler at the heart of Chromium's compositor. On the GitHub mirror of Chromium (`chromium/chromium`, measured 17 November 2026), the 400 most recent commits to `cc/scheduler/scheduler.cc` run from 10 April 2014 to 16 November 2026 (newest `2de4fbaf3e`), and 48 of their subjects name BeginFrame. Twelve years of change, and the contract that a frame starts when the display compositor says so is still the core of the file.

The Frame Ledger method

The Reader's Brief showed a filled Ledger. Here is how a team produces one. As a rule of thumb, not a measured figure, plan about a week for the first screen and about a day for each one after.

- **1. Pick one critical screen and one reference device per OS.** Critical means money, consent, safety or a regulator: Halbrook's payment approval, Corvane's kiosk check-in, Larkfield's sign-in. The reference device is the oldest or weakest model still in your fleet, not the developer's laptop.
- **2. Trace one frame with the platform tool.** Use the tools above. For each hop, write the process that ran it, the time it took and, where the tool shows it, the memory. Repeat the trace enough times to see the slow frames, not only the typical one.
- **3. Fill owner, payer and channel.** Take them from vendor documentation and from your Pixel Bill of Materials. Write the version policy: evergreen, pinned, OS-tied or bundled.

- **4. Mark untrusted inputs and readers.** Which hops take content from outside, such as remote fonts, images or third-party scripts? Who can read each hop's output: capture tools, agents, assistive technology, a remote-display client?
- **5. Measure the patch delay.** For the last three serious flaws fixed in that hop's component, count the days from the upstream fix to the fix running on your reference device. For an evergreen runtime, count to the app restart, not to the download.
- **6. Decide and assign.** Write Own, Rent, Pin or Isolate per row, with one line of reason, and name one accountable person per row.

Three rules keep the Ledger honest. First, every number has a source: a trace file, a vendor page, a commit. Second, "not observed" and "untested" are valid entries; a blank is not. Third, the Ledger records the path the screen really takes on the reference device. If a kiosk's GPU is blocked and the engine falls back to software raster, the Ledger says so, even though the design document says GPU.

THE UNSEEN ANGLE

A frame can be perfectly smooth and still late. Perfetto's FrameTimeline documentation has a category for frames that "are presented late, resulting in an increased input latency" while the frame rate stays steady, and it describes "BufferStuffing", where an app queues frames faster than they can be shown, as "more of a state than a jank". A frame-rate dashboard reads 60 and stays green. The user feels every tap arrive late. Only a trace that times the frame to its presentation, not to the end of your code, can see the difference. That is why the Ledger's extended sheet has an "Observable by" column, and why "nothing" is a common answer in it.

> *Your code can finish on time and the frame can still be late. The deadline belongs to the display, and the delay belongs to whichever owner held the frame last.*

This hop in the Frame Ledger

Corvane's self check-in kiosk runs on Windows IoT and shows a web page in WebView2, here in its Fixed Version mode: Corvane packages a pinned copy of the runtime with its kiosk app, instead of the Evergreen copy that updates itself. These are the rows that decide whether a guest's tap feels instant.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision
Input side row: kiosk touch controller and Windows pointer input	Kiosk hardware maker for firmware; Microsoft for Windows; Corvane pays through purchase and licensing	Windows servicing on the kiosk image's channel; touch firmware rarely	Kernel and the window's owning thread	Windows; any tool allowed to inject or observe input	**RENT:** measure tap-to-light on the reference kiosk at every image release
0 to 5 Renderer main thread: Blink runs the check-in page's script, style, layout and paint	Corvane writes the page; Chromium upstream writes Blink; Microsoft ships the runtime	The page: Corvane. The runtime: Corvane, when it repackages a newer Fixed Version; Microsoft's distribution guide says "the most-patched version of the latest and second-latest major releases" is available to download	Renderer process, Untrusted integrity per Chromium's sandbox design	Page scripts; DevTools if left enabled	**OWN, PIN:** own the page, with a main-thread budget per frame; pin the runtime, with a written repackaging deadline
6 Raster: Chromium raster with Skia, on the kiosk GPU or, on a blocklisted GPU, on the CPU	Chromium and Skia upstream; Microsoft ships; the GPU maker for the driver	As the runtime row; the driver with Corvane's kiosk image	GPU process (Low integrity) for GPU raster; CPU raster workers when the GPU is not trusted	Not the page	**PIN, RENT:** pinned with the runtime; record the raster mode per kiosk model in the trace (Chapter Four)
7 Engine compositor: Chromium cc scheduler and Viz	Chromium upstream; Microsoft ships	As the runtime row	Renderer compositor thread and GPU process (Low integrity)	Not the page	**PIN, ISOLATE:** pinned with the runtime; keep the GPU sandbox on
8 OS compositor: DWM presentation	Microsoft	Windows servicing, monthly security releases	`dwm.exe`	Windows capture APIs and any tool that uses them	**RENT:** trace with WPR's desktop composition profile before each image release
9 to 10 Scanout, display, glass and readers: kiosk panel, run at a fixed 60 Hz	Panel maker; Corvane buys it	Panel firmware, rarely updated	None	Guests, staff, cameras in the lobby	**OWN** the policy: an approved-panel list and a fixed refresh rate, so the deadline is known
Capture side row: remote-support and screen-capture tools in the kiosk image	Each tool's vendor; Corvane pays	Each vendor's own channel	Each tool's own process	Whoever the tool streams or records to	**ISOLATE:** list the allowed tools in the kiosk image; where the trace shows no capture, write "not observed", not a blank

SECURITY LENS

Who can change it? Anyone who adds work to the UI thread can move a frame past its deadline: your own developers, every third-party script on the page, and any tool that hooks the window. A UI thread that is stuck cannot repaint, so a warning, a consent prompt or a changed amount may not appear when the user acts. Below your code, the runtime vendor, the GPU maker and the OS vendor each change the presentation path through their own updates.

How fast can it be fixed? A frame regression in your code is fixed on your release train. One in the engine arrives on the engine's train: Chrome moved to a new major release every four weeks starting with Chrome 94 in September 2021, and WebView2's evergreen runtime receives Edge's updates, active only when the app restarts. A pinned runtime, like Corvane's, is fixed when Corvane repackages it. A panel is often never fixed.

What does AI change here? Our view (as of November 2026): AI coding assistants make it cheaper to add script and components to a screen, so main-thread budgets need an owner who says no. Agents that read the screen act on the last presented frame, which may lag the app's real state by the pipeline depth this chapter measured; Chapter Ten treats that gap.

BUSINESS IMPLICATIONS

Cost. For Corvane, a kiosk that drops frames on a weak GPU or a fixed-version runtime that nobody repackages is paid for later, in lobby queues and in site visits. A traced reference kiosk at each image release costs a day and finds both before they ship to about 3,000 kiosks across 300 properties.

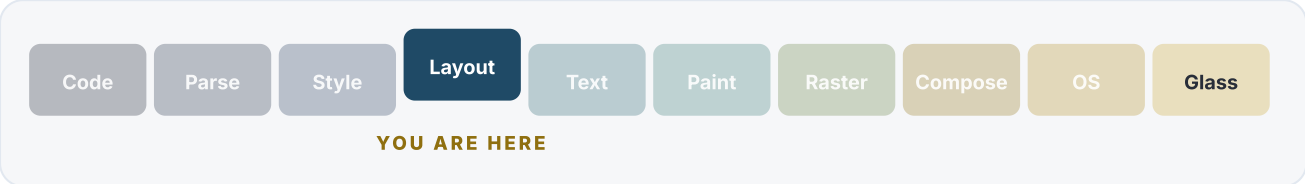
Visibility. Larkfield's monitoring times its own functions, which ends where the frame leaves the main thread. Add one platform trace per release for its slowest view, and record in the Ledger which hops are "not observed". The list itself is the finding.

Value. What Halbrook's tellers call "fast" is input-to-photon latency and steady frames, not a benchmark score. A screen that answers every tap within a couple of intervals earns trust that no feature list buys.

Monday rule: pick your most critical screen, trace one frame on your weakest supported device, and write down which owner held the frame when it was late.

CHAPTER TWO

From Markup to Geometry: Parse, Style and Layout



Before a single pixel is drawn, the engine has already turned your code into a set of trees, three of them in the simplified model this chapter uses, and a page of geometry, and it redoes part of that work every time something changes. The decision this chapter changes is how you budget a screen: not by how much data it shows, but by how much markup, style and geometry the engine must rebuild per change, and by whose layout code does the rebuilding. A dashboard that feels slow on half your customers' machines is usually losing its frame here, long before the GPU sees it. And the layout code under your product belongs to one of three engine families you did not write.

Three trees before the first pixel, simplified

A web engine does not draw HTML. It builds data structures from it, and each stage feeds the next. The three trees below are a simplified model: Blink, for example, also keeps the layout objects that produce its fragments, and paint property trees (transform, clip, effect and scroll) that describe how visual effects apply, as its RenderingNG documentation lists.

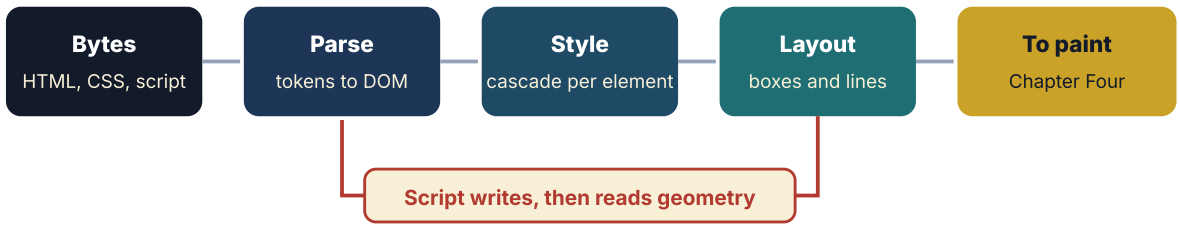
- **Parse** turns bytes into a tree of elements, the **DOM** (Document Object Model). Scripts read and change this tree.
- **Style** works out, for each element, the final value of every CSS property after all the rules that match it have been combined. The result is the **computed style**.
- **Layout** turns elements and their computed style into geometry: the position and size of every box, line and glyph run. Blink, the engine in Chrome and Edge, calls its output a **fragment tree**.

Native toolkits follow the same idea under other names. A WinUI app on Windows builds a XAML visual tree. A SwiftUI or UIKit app on Apple platforms builds a view tree. Jetpack Compose on Android builds a composition tree, and Qt Quick on Linux builds a scene graph.

Each one keeps a model of the screen, works out sizes and positions from it, and hands the result to paint. The vocabulary changes; the cost structure does not. A change to the model costs work in proportion to how much of the model it invalidates.

FIGURE 2.1

From bytes to geometry, and the loop that breaks the budget



Normal path: style and layout run once per frame, for the parts that changed.
Forced path: each read after a write makes the engine run style and layout now.
Inside a loop, that repeats per item: the pattern called layout thrashing.

A simplified model: the trees are rebuilt incrementally. The red loop is how ordinary application code turns an incremental update into a full recalculation, many times inside one frame.

Parsing: a specified algorithm that forgives

HTML parsing is not left to each vendor's taste. The WHATWG HTML standard defines it as an algorithm: the input "is passed through a tokenization stage followed by a tree construction stage", and the output is a document. The standard also says that "the error handling for parse errors is well-defined". Broken markup is not rejected; every conforming engine is told exactly how to repair it, so that the same bad page produces the same tree everywhere.

That design is why the web works, and it is also why the parser matters to a CISO. The parser is the first code that gives untrusted bytes an HTML structure. Network code, the MIME type check and character decoding touch the bytes before it, but the parser is the first to act on what the markup means. It runs in the renderer process, the sandboxed process that handles web content in Chromium-based engines. Its job is to accept anything. Two consequences follow for an enterprise screen:

- Markup that your own code builds from data, such as a customer name or a note field, goes through the same forgiving repair rules. If you build HTML by joining strings, the parser will repair it into a tree you did not intend. Build the tree through the DOM or a framework that escapes values.

- A classic script that the parser itself inserts, written as a plain script tag without the `async` or `defer` attribute, pauses parsing while it loads and runs, because the script may change the document being built. Heavy scripts of that kind at the top of a page delay every later stage of the frame.

Style: the cascade, and the CSS you did not write

The cascade is the rule set that decides which of several matching CSS rules wins for each property on each element. It dates to CSS level 1, a W3C Recommendation of 17 December 1996, and every engine still implements it. Engines make it fast with **style invalidation**: when a class or attribute changes, they work out which elements could be affected and recompute only those.

Invalidation only works when the engine can tell what is affected. Broad selectors, deep nesting and rules that depend on siblings or descendants make the affected set larger. On an enterprise dashboard the worst offenders are often not your own stylesheets. A charting widget, a data grid, an analytics tag or a chat panel brings its own CSS, written by another vendor, and the engine treats it like yours. When that vendor ships a new version, your style cost changes on the day your users load it.

Engines have also changed how style is computed. Mozilla rewrote Firefox's style system in Rust, taking it from the Servo research engine. Mozilla's documentation states: "Starting with Firefox 57 and later, Gecko makes use of the parallel style system written in Rust that comes from Servo." Style for different elements can run on several cores at once. Parallel style makes a large style recalculation cheaper; it does not make an unnecessary one free.

Layout: geometry, and the read that forces it

Layout computes where everything goes. It is the most algorithmically complex of the three stages, because CSS has several layout models, including block, inline text, tables, flexbox and grid, and each can nest inside the others. Text layout depends on the font and shaping results of Chapter Three, which is why a font change can move a button. Layout and text work as a loop, not a hand-off: hops 3 and 4 run together, because layout asks the text stack to shape runs to find line breaks. Blink's `ShapingLineBreaker` finds the ideal break from the shape results for the paragraph and reshapes the start or end of a line when the break does not fall at a boundary HarfBuzz marks as safe.

Engines batch layout. A script can change many styles, and the engine lays out once, just before the frame is painted. That changes when a script asks for geometry after changing something. Google's web.dev guidance describes it plainly: reading a value such as `offsetHeight` after a style change makes the browser run style and layout early, a **forced**

synchronous layout. Doing that repeatedly in a loop, read, write, read, write, is **layout thrashing**, and the cost grows with the size of the DOM. The fix is to batch all reads before all writes.

This is visible if you look. Chrome DevTools has a "Forced Reflow" insight that names the function responsible, and the Long Animation Frames API exposes a `forcedStyleAndLayoutDuration` value, so the same problem can be measured in the field, on customers' machines, not only on a developer's workstation.

Three layout families, and how new their code is

Almost every screen a company ships on the web platform is laid out by one of three engine families: Blink (Chrome, Edge, Electron, CEF and Android's System WebView), WebKit (Safari and every WKWebView on Apple platforms) and Gecko (Firefox). It has been three since January 2020; Microsoft's own engine, EdgeHTML, was a fourth until the new Edge, built on Chromium, shipped on 15 January 2020, and Microsoft ended support for the legacy Edge desktop app on 9 March 2021.

A common story says this code is old: the KHTML engine from KDE, written more than twenty years ago, still lays out the web. The lineage is real. The code is newer than the story. Google's Chrome team states that "as of Chrome 108, the legacy engine is no longer used to perform layout", after a multi-year rewrite called LayoutNG whose first parts shipped in 2019. WebKit is building a new layout system of its own, Layout Formatting Contexts, under `Source/WebCore/layout/`, and integrating it step by step with its older rendering tree. Servo, the Rust engine, replaced its first layout engine entirely; the box below has the dates from its source history.

For a CTO, the lesson is not nostalgia. Lineage tells you which family's behavior and bugs you inherit. Rewrites tell you that layout behavior changes underneath a stable brand name. A layout engine rewrite is a period when long-standing quirks are fixed and new regressions appear, and the version of the engine you ship decides which side of that change your users are on.

Layer	Windows	macOS	iOS	Linux	Android
Native scene model	XAML visual tree (WinUI)	SwiftUI or AppKit view tree	SwiftUI or UIKit view tree	GTK widget tree, Qt Quick scene graph	View hierarchy, Compose tree
Layout family in the usual embedded web view	Blink, in Microsoft's Edge WebView2 runtime (WebView2)	WebKit (WKWebView)	WebKit (WKWebView), in every ordinary app	No single default: WebKit (WebKitGTK) or Blink (Qt WebEngine, CEF)	Blink (Android System WebView)
Who updates that layout code	Microsoft, through WebView2	Apple, with OS and Safari updates	Apple, with iOS updates	The distribution, or you if you bundle it	Google Play updates of the WebView, where Play is present

The table changes one decision. A cross-platform product that uses each OS's web view is laid out by two families at once, Blink on Windows and Android, WebKit on Apple platforms, and must test both. A product that bundles its own engine gets one family everywhere except iOS, where WebKit is required unless Apple grants an alternative-engine entitlement, offered only in the European Union and Japan; and it takes on the job of updating that engine.

HOW IT GOT THIS WAY

The cascade and the box model come from CSS level 1, which the W3C issued as a Recommendation on 17 December 1996. On 7 January 2003 Apple announced Safari and described its rendering engine as "based on KHTML, from KDE's Konqueror open source project". In June 2005 Apple opened the wider WebKit project, with public source access and an open bug database. On 3 April 2013 Google announced Blink, a fork of WebKit for Chromium. Gecko comes from the Netscape and Mozilla line. Those forks are why three families exist and why their quirks differ.

Each family then rebuilt its layout core. Blink finished LayoutNG in Chrome 108. Firefox took Servo's parallel style system in Firefox 57. Servo's own history shows how long such a rewrite takes. In the Servo repository, commit `2b01c26aa5` of 27 June 2019 added a build option for "layout 2020". Servo's team recommended it in April 2023 as "the best layout engine for Servo going forward". Commit `7594dc6991` of 13 March 2025, titled "Remove legacy layout (layout 2013)", deleted the old engine. That is almost six years from first switch to removal, for a team that started with no web compatibility debt to protect.

What hasn't changed: the CSS cascade and box model of 1996 still define the input to every one of these engines; the layout code that implements them has been rewritten (Servo repository, `components/layout_2013`, removed in commit `7594dc6991`, 13 March 2025; measured 18 November 2026).

THE UNSEEN ANGLE

Vendor brands are more stable than vendor code. "It is Chromium" or "it is WebKit" sounds like one fixed thing, but the layout engine under each name was substantially rewritten in the last decade, and the rewrites shipped feature by feature: Chrome's team lists fragmentation in Chrome 102, flex and grid fragmentation in 103, table fragmentation in 106 and printing in 108. A team that pins an engine version for stability is also pinning a point in the middle of someone else's rewrite. Record the engine version per screen in the Ledger, not only the brand, because the version decides which layout code your customer is running. (Sources: Chrome for Developers, "RenderingNG deep-dive: LayoutNG block fragmentation"; Servo repository history.)

> *You do not choose a layout engine once. You choose it every time you pick an engine version, and the version decides whose rewrite your customer is living in.*

The cost of a change

Larkfield Software's analytics view shows a grid of several thousand cells, a chart and a filter panel, all in its bundled Chromium-based engine. A user changes one filter. What that costs depends on design choices most product teams never write down:

- **How much of the DOM the change touches.** Replacing the whole grid rebuilds DOM, style and layout for thousands of nodes. Updating the changed cells touches a few.
- **Whether geometry is read in a loop.** A grid that measures each row after inserting it forces layout once per row.
- **Whether layout can be contained.** CSS containment and fixed-size containers tell the engine a change inside a box cannot move anything outside it, so less of the page is laid out again.
- **Whose CSS is in the page.** Each third-party widget adds selectors that every style recalculation must consider.

None of this appears in server logs or application performance monitoring that stops at the network. It appears in a browser trace, as long style and layout tasks on the main thread, and it varies by customer machine. Larkfield's support team hears "the app freezes when I filter"; the trace says layout thrashing in a grid component the team bought.

This hop in the Frame Ledger

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
1 Parse: Blink HTML parser in Larkfield's bundled engine	Chromium project, largely Google; Larkfield pays to ship and test each copy	Larkfield, only when it ships an engine upgrade to 400,000 desktops	Renderer process, sandboxed when the sandbox is left on	Page scripts and any loaded third-party script	**RENT, ISOLATE:** rent the parser; isolate any remote HTML in a separate sandboxed view; never build markup by joining strings
2 Style: Blink style engine plus third-party widget CSS	Chromium project; widget vendors write their own CSS; Larkfield pays the CPU cost on customer machines	Engine: with Larkfield's upgrades. Widget CSS: whenever the vendor ships	Renderer process	Page scripts (computed style is readable)	**RENT, OWN, PIN:** rent the engine; own a style budget per hot view; pin widget versions
3a Layout: Blink LayoutNG in the desktop app	Chromium project, largely Google	Larkfield, per engine major it adopts; regressions arrive with upgrades	Renderer process	Page scripts through geometry APIs; assistive tech through the accessibility tree	**RENT, PIN:** pin per release; run layout regression tests on the top screens before each upgrade
3b Layout: companion iOS app, SwiftUI and WKWebView	Apple	Apple, with iOS updates; Larkfield cannot ship its own engine there	App process; WebKit web content process for web views	The app; assistive tech	**RENT:** test the WebKit family separately from Blink

SECURITY LENS

Who can change it? The engine project writes the parser, style and layout code. The company that bundles the engine decides which version runs. Any script or stylesheet loaded into the page, including a third-party widget, changes the tree and its cost at run time.

How fast can it be fixed? For an engine supplied with the OS or an evergreen runtime, as fast as that vendor's update reaches the device. For a bundled engine, only as fast as the product team ships its next release and customers install it. A parser fix upstream does nothing for a copy nobody upgraded.

What does AI change here? Our view (as of November 2026): AI coding assistants already write a growing share of the markup and CSS in new screens, and that code is easy to accept on sight, so style and layout cost can grow without anyone choosing it. Agents that read the DOM or call geometry APIs also become readers of this hop, which makes the DOM a second interface to the screen that needs its own review (Chapter Ten).

BUSINESS IMPLICATIONS

Cost. Style and layout run on the customer's CPU, not yours, so their cost is invisible in your cloud bill and visible in support calls and battery complaints. For Larkfield, a forced-layout loop in one grid view is multiplied by every desktop that opens it. For Corvane Hotels, the same loop on a low-power kiosk is the difference between a responsive check-in screen and a guest pressing a button twice.

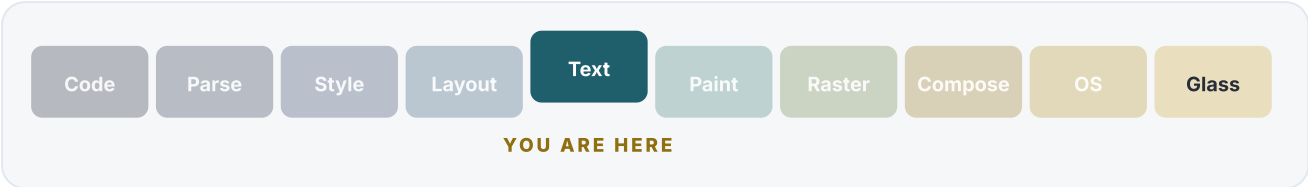
Visibility. Server logs and most application monitoring end before this hop. Halbrook Bank's teller app can report healthy API times while its approval screen spends hundreds of milliseconds recalculating style. Field data from the Long Animation Frames API, or traces from DevTools and Perfetto on reference machines, is the only witness.

Value. On Windows and Android, the usual engines, from Chrome and Edge to WebView2, Electron, CEF and the System WebView, lay out with Blink, so on a Windows or Android fleet without Firefox one layout family draws every web screen. A regression in it can reach all those customers on the same day, and so can a fix. Knowing which family and which version renders each critical screen turns a vague "the browser changed" into a dated, testable item in the release plan.

Monday rule: for your three busiest screens, record the engine family and version, the DOM size, and the longest style and layout task in a field trace; set a budget for each and fail the build when a change exceeds it.

CHAPTER THREE

Text: Shaping, Fonts and the Glyph



The most important pixels on a regulated screen are letters: an amount, a payee, a consent sentence. The decision this chapter changes is who you let supply the fonts and characters on those screens, and whether you trust a string because it looks right. A font file is active, attacker-controlled input that has reached the Windows kernel before. A payee name can display one way and be stored another. And the brand font your marketing team licensed may not be licensed for the app you are about to ship it in.

Who shapes, who draws, on each system

Text on a screen is four problems. **Encoding** decides which characters a string holds, as Unicode code points. **Fonts** are the files that hold the drawable shapes and the rules for using them. **Shaping** turns a run of characters into positioned glyphs, the drawable shapes in a font: it joins Arabic letters, reorders Indic vowel signs and places accents. **Rasterization** turns each glyph's outline into pixels at a given size. **Fallback** is what happens when the chosen font has no glyph for a character, and the engine must find another font that does. This chapter follows those problems through the frame.

The owners differ by system and by engine, and the difference matters when something breaks:

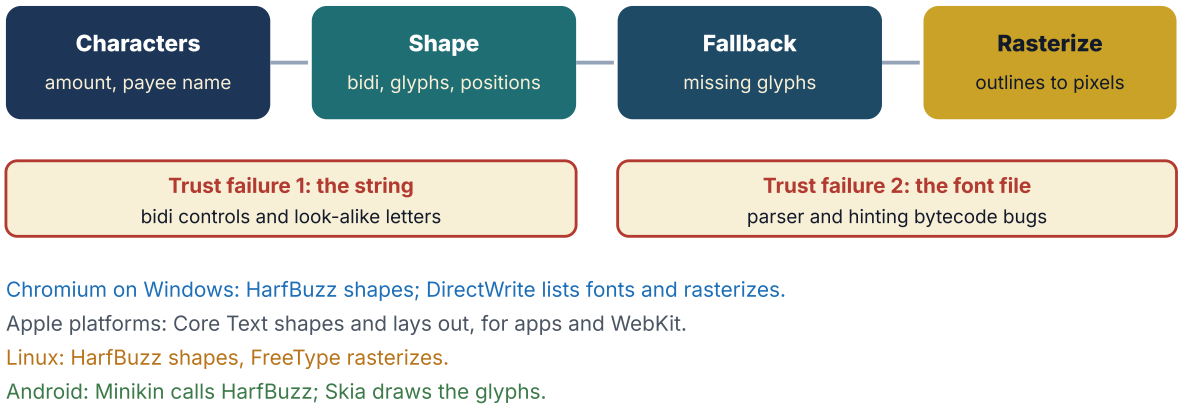
- **Chromium-based engines**, including Chrome, Edge, Microsoft's Edge WebView2 runtime (WebView2), Electron and CEF, shape text with **HarfBuzz** on every platform, including Windows. Blink's text code lives in `shaping/harfbuzz_shaper.cc`. On Windows, DirectWrite supplies the system font list, font metrics and glyph rasterization. Because the sandbox "prevents the renderer from using the system font collection", in the words of Chromium's design note, the sandboxed renderer asks the browser process over IPC for each system font and each fallback font it needs.
- **Native Windows apps** use DirectWrite for shaping and rasterizing; older GDI apps use GDI's text path.

- **Apple platforms** use Core Text to shape and lay out text, for native apps and for WebKit.
- **Linux desktops** use HarfBuzz to shape and FreeType to rasterize.
- **Android** lays out text with its Minikin library, which calls HarfBuzz to shape; Skia draws the glyphs.

So one Halbrook Bank payment screen, opened in the teller app's WebView2 control and in the bank's iOS app, is shaped by two different engines, with two different fallback lists, and rasterized by two different rasterizers. The same string can wrap differently in each.

FIGURE 3.1

One string, three owners, two trust failures



Chromium on Windows: HarfBuzz shapes; DirectWrite lists fonts and rasterizes.
Apple platforms: Core Text shapes and lays out, for apps and WebKit.
Linux: HarfBuzz shapes, FreeType rasterizes.
Android: Minikin calls HarfBuzz; Skia draws the glyphs.

The string can be false before any font is touched, and the font can be hostile before any glyph is drawn. Both failures sit upstream of every compositor in this book.

The font file is active input

A TrueType font is not just pictures of letters. It carries **hinting instructions**: small programs, written in a stack-based bytecode, that adjust glyph outlines to fit the pixel grid at small sizes. Microsoft's OpenType specification documents the TrueType instruction set and its stack. The bytecode runs in a constrained interpreter inside the rasterizer, not as native machine code. That is the right way to describe the risk: a font is active, attacker-controlled input to a complex parser and an interpreter, and both have had bugs that let an attacker write memory.

The record is specific:

- **CVE-2011-3402.** NVD describes a flaw "in the TrueType font parsing engine in win32k.sys", the Windows kernel-mode graphics driver, triggered by "crafted font data in a Word document or web page", and exploited in the wild in November 2011 by the Duqu malware. Because the parser ran in the kernel, a document could reach kernel code. CISA added the flaw to its Known Exploited Vulnerabilities (KEV) catalog on 6 October 2025, fourteen years later.
- **CVE-2020-15999.** A heap buffer overflow in FreeType, fixed in Chrome 86.0.4240.111, reachable through "a crafted HTML page". CISA added it to KEV on 3 November 2021.
- **CVE-2025-27363.** An out-of-bounds write in FreeType 2.13.0 and earlier, while parsing variable font structures; newer FreeType releases were not affected. CISA added it to KEV on 6 May 2025. The exposure lived in every product still carrying an old copy.

Operating systems and engines responded by moving font parsing to places where a bug does less damage. Microsoft's security baseline team wrote that "with Windows 10, GDI font parsing is no longer performed in kernel mode"; it runs in `fontdrvhost.exe`, "a highly-restricted, per-session AppContainer process". Microsoft's advisory ADV200006 adds the version detail: in early Windows 10 releases installed fonts were still processed in kernel mode, and from Windows 10 version 1703 "all fonts are processed in fontdrvhost.exe in user mode appcontainer". Chromium checks web fonts with the OpenType Sanitizer (OTS) before use, and parses them inside the sandboxed renderer. Since 2024 it has also been replacing FreeType with **Fontations**, a family of Rust font libraries.

Fallback, wrapping and the glyph nobody tested

Fallback is where text bugs hide, because it only runs for some characters. A name written in Latin letters uses the bank's brand font. The same field holding an Arabic, Thai or Chinese name falls back to a system font with different widths, different line height and different vertical metrics. The line wraps, a button moves below the fold, and the bug appears only for customers with names in those scripts.

Rasterization differs as well. Apple's own engine code records that "macOS Mojave disabled text subpixel antialiasing by default" (WebKit changeset r239306, December 2018). Subpixel antialiasing uses the red, green and blue stripes of an LCD pixel to smooth letter edges. After Mojave, the same font at the same size looks lighter on a Mac than on a Windows machine using ClearType, and differs again under FreeType on Linux. A screenshot taken on one system is not evidence of how text looked on another, a point Chapter Nine returns to.

Text job	Windows	macOS	iOS	Linux	Android
Shaping in native apps	DirectWrite	Core Text	Core Text	HarfBuzz	HarfBuzz, through Minikin
Shaping in the usual embedded web view	HarfBuzz (Blink in WebView2)	Core Text (WebKit)	Core Text (WebKit)	HarfBuzz (WebKitGTK or a Blink-based view)	HarfBuzz (Blink in System WebView)
Glyph rasterization	DirectWrite	Core Text and Core Graphics	Core Text and Core Graphics	FreeType	Skia, with FreeType or Fontations in Chromium
Where an untrusted font is parsed	GDI fonts: `fontdrvhost.exe` AppContainer. Web fonts: sandboxed renderer	Web fonts: WebKit web content process	Web fonts: WebKit web content process	Native apps: usually the app process	Native apps: the app process. Web fonts: sandboxed renderer

The last row is the decision. On Linux and Android, a native app that loads a font from the network parses it in its own process, with the app's privileges. On Android the app sandbox still confines that process, so a parser bug reaches the app's own data and permissions, not other apps. A desktop Linux app outside Flatpak or a similar sandbox has no such wall: the bug runs with the user's full rights. In both cases no separate parser sandbox protects the app's own data, and turning off remote fonts removes that attack surface.

Text that looks right and is false

Unicode text can be stored in one order and displayed in another. The Unicode Bidirectional Algorithm, Unicode Standard Annex #9 (revision 51, dated 13 August 2025, part of Unicode 17.0, released 9 September 2025; revision 52, for Unicode 18.0, is dated 1 September 2026), decides the display order of mixed left-to-right and right-to-left text. It includes invisible control characters that override direction. In 2021 researchers showed that these controls could make source code display differently from what a compiler reads; NVD records it as CVE-2021-42574, an issue that "permits the visual reordering of characters via control sequences".

The same mechanism works on a payee field. A name can contain direction controls that reorder an account reference on screen, or look-alike letters from another script, called **confusables**, that make a different name look identical to a trusted one. Unicode's security guidance began in Unicode Technical Report #36, "Unicode Security Considerations". Its banner now reads "Stabilized Technical Report": it "has not been maintained since 2014", and some of its material "has been superseded by" UTS #39, "Unicode Security Mechanisms", which defines confusable detection, and UTS #55 for source code. A screen can render the font perfectly and still show the wrong thing, because the string was false before shaping began.

The font is also a contract

Fonts are licensed software. The OpenType specification gives each font an `fsType` field that declares embedding permissions, such as "Restricted License embedding" or "Preview & Print embedding", and states that "embedding licensing rights are granted by the vendor of the font". The field describes document embedding; the license agreement is the controlling document.

Bundling a font file inside an Electron, CEF or WebView2 app is a different use from installing it on employees' desktops. Foundries sell it separately: Monotype, for example, offers an "App License" for embedding a font file in a mobile application's code, apart from its desktop license. A brand font licensed for marketing desktops is not automatically licensed for 400,000 copies of Larkfield's installer. The procurement question is simple: does our license cover embedding this font in a distributed application, on how many installs, and for how long?

HOW IT GOT THIS WAY

Apple "included full TrueType support in its Macintosh operating system, System 7, in May 1991", in Microsoft's own history of the format, and Microsoft "first included TrueType in Windows 3.1, in April 1992". The hinting bytecode came with it. David Turner created FreeType in 1996 to render TrueType fonts, and FreeType 2.0, a modular rewrite, followed in 2000. For speed, Windows parsed fonts inside the kernel graphics driver, which is how a font in a Word document reached kernel code in 2011.

The response took a decade. Windows 10 moved GDI font parsing into `fontdrvhost.exe`, and from version 1703 GDI processing of all fonts, installed ones included, went there. Chrome began using Fontations in Chrome 128 (August 2024) for less common formats such as color fonts, and from Chrome 133 (February 2025) for all web fonts on Linux, Android and ChromeOS, and as a fallback on Windows and Mac. Google's post (Chrome for Developers, "Memory safety for web fonts", 19 March 2025, with a later update note) adds that Fontations took over system fonts on Linux and Android in Chrome 136 and that FreeType was removed from Blink in Chrome 145.

What hasn't changed: the TrueType instruction set that shipped with System 7 in May 1991 is still part of the OpenType specification today, and every TrueType-capable rasterizer, including the new Rust ones, still has to interpret it. FreeType's interpreter, `src/truetype/ttinterp.c`, was already in the project repository's first commit (`d2b1f357`, "Initial revision", David Turner, 16 December 1999); 448 commits have changed it since, and it is 7,631 lines long today (FreeType repository at `d7339729`, measured 18 November 2026).

THE UNSEEN ANGLE

"What you see is what you sign" usually fails long before the compositor. A payment approval screen can be correctly composed, correctly captured and correctly recorded, and still show a payee that differs from the stored one, because direction controls or confusable letters were in the string. No rendering control downstream can detect that; only a check on the characters, before they are shaped, can. The standards for that check exist and are specific: UAX #9 for direction, UTS #39 for confusables. Whether a payment screen applies them to payee names is a design choice that shows up in no rendering trace, so ask for it by name. (Sources: Unicode UAX #9, UTS #39, UTR #36 status note; NVD CVE-2021-42574.)

> *A payee name can be drawn perfectly and still be false. Check the characters before you trust the glyphs.*

This hop in the Frame Ledger

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
0 Source: payee string from the payment record	Halbrook's payments team	Halbrook, at its release speed	Bank back end, before rendering	Every reader of the screen	**OWN:** reject or flag direction controls and mixed-script confusables (UTS #39) before display
4a Text shaping: HarfBuzz in Halbrook's WebView2 teller app	HarfBuzz project; Chromium upstream; Microsoft ships the runtime	Microsoft, through the evergreen runtime; active only after the app releases the old runtime or restarts	Renderer process, sandboxed	Page scripts; accessibility tree; screen agents read the text	**RENT:** the shaping code comes with the runtime; plan app restarts so fixes become active
4b Font file: Halbrook's licensed brand font	Foundry writes it; Halbrook pays the license	Halbrook, when it replaces the file	Checked by OTS, parsed in the sandboxed renderer	Anyone who unpacks the app	**OWN:** confirm app-embedding rights; ship it in the app package; block remote fonts on the approval screen with a `font-src` content security policy
4c Fallback: DirectWrite system fonts	Microsoft	Windows Update, monthly	Browser process, through the font proxy	n/a	**RENT:** test the approval screen with Arabic, Chinese and Thai payee names
4d Legacy GDI text: older components on branch PCs	Microsoft	Windows Update, monthly	`fontdrvhost.exe`, AppContainer	n/a	**RENT:** confirm no branch PC runs a build older than Windows 10 1703, where installed fonts were still parsed in the kernel

SECURITY LENS

Who can change it? Anyone who can supply a font or a string to the screen: a web page with a remote font, a document with an embedded one, a payee who chose their own name. The font engine's maintainers fix parser bugs; the OS or runtime vendor decides when the fix ships; each app that bundles a copy decides when its users get it.

How fast can it be fixed? For fonts parsed by the OS, as fast as monthly OS updates. For a bundled FreeType or a pinned engine, as fast as the slowest product in the fleet. CVE-2011-3402 entering the KEV catalog in October 2025 shows how long old parsers stay in service somewhere.

What does AI change here? Font parsers are a natural target for automated fuzzing, and Google wrote in "Memory safety for web fonts" (19 March 2025) that keeping up with FreeType fuzzing reports cost it "at least 0.25 full time software engineers". AI-assisted fuzzing raises the rate of findings against the copies you bundle. Screen-reading agents also read the shaped text, so a confusable payee fools an agent that reads pixels as easily as it fools a teller.

BUSINESS IMPLICATIONS

Cost. Fonts carry license costs per embedding and per install, and they are easy to miss. Larkfield needs app-embedding rights for any brand font in its installer; Corvane needs the same for fonts on 3,000 kiosks and its Android tablets. Budget the license review alongside the font, not after the audit letter.

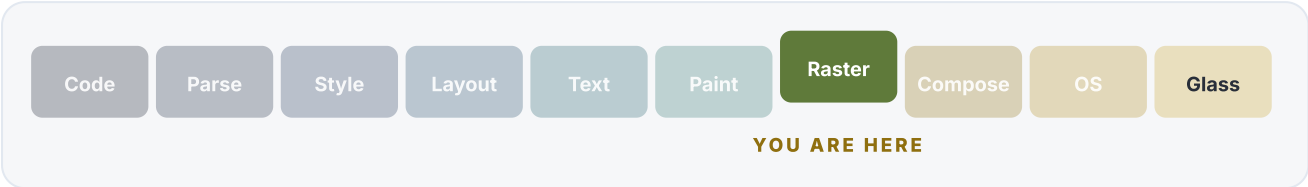
Visibility. Text failures appear for some scripts, some systems and some customers only. Halbrook Bank will not see a fallback wrap or a confusable payee in its dashboards. A test set of real names in the scripts its customers use, rendered on each target system, is the only way to see them before customers do.

Value. Correct text in every script is market access, and correct characters on payment screens are fraud prevention. Halbrook can turn a check on payee strings into a control it can show an auditor, which is worth more than any claim about the pixels.

Monday rule: on every regulated screen, block remote fonts, ship only licensed fonts in the app package, and check amount and payee strings for direction controls and confusables before they are rendered.

CHAPTER FOUR

Paint and Raster: From Display List to Pixels



Somewhere in every frame, real pixels are computed, on a GPU if the engine trusts your hardware and on the CPU if it does not. The decision this chapter changes is how you qualify devices: not by whether the app runs, but by whether it rasters on the GPU, at what resolution, and with how much memory to spare. That choice is often made for you by a list of drivers an engine vendor refuses to trust, and when it goes the wrong way nothing crashes. The screen just gets slower, hotter and more expensive.

Paint records; raster executes

After layout, the engine knows where everything goes. It does not draw yet. **Paint** walks the layout result and records drawing commands: fill this rectangle, draw these glyphs at these positions, draw this image with this clip. The recording is a **display list**. Chromium calls it a `PaintRecord`, which its documentation describes as "the SkPicture equivalent that stores a number of PaintOps".

Raster is the step that executes those commands and produces pixels. Separating the two has practical benefits. A display list is small and cheap to keep; pixels are large. A list can be replayed at a new zoom level without repainting, sent to another process, or rastered in pieces on several threads. Native stacks use the same split. Android's HWUI records drawing on the app's main thread and replays it on a separate render thread, with Skia doing the raster. Apple's Core Animation keeps each layer's content separately so that unchanged layers do not need to be drawn again.

For a CTO, the split explains a common surprise. Paint is usually cheap and raster is usually expensive, and they run in different places, often different processes. A trace that shows a short paint step has not shown you the cost of the frame.

Tiles, threads and the GPU

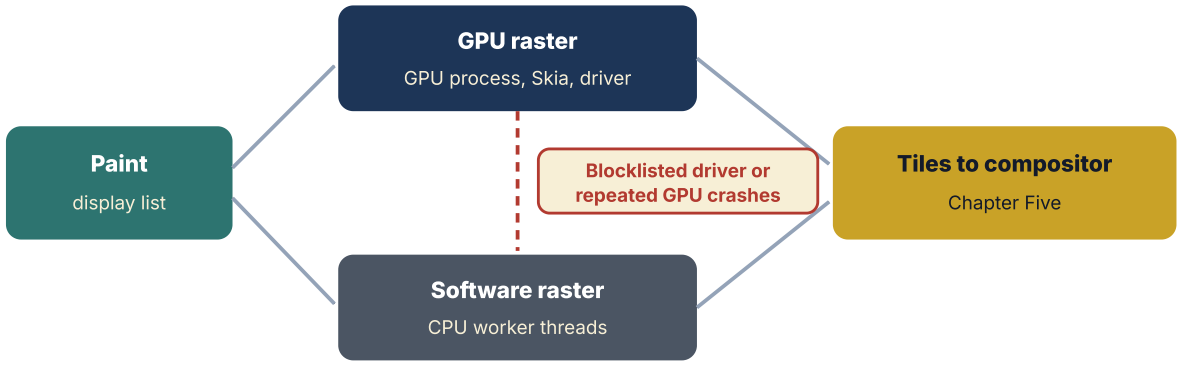
Chromium's compositor, `cc`, does not raster a whole page at once. It cuts each layer into **tiles** and rasters the ones that are visible or about to be. Its documentation gives the sizes: "for software raster tiles are roughly 256x256 px and for gpu raster tiles are roughly viewport width x one quarter viewport height". A component called the TileManager decides which tiles to raster first, and schedules the work across worker threads.

There are two raster modes. In **software raster**, worker threads produce bitmaps on the CPU. In **GPU raster**, the engine sends paint commands over a command buffer to the GPU process, which turns them into textures on the graphics card. The same documentation notes two limits worth knowing. Image decoding is "the most expensive part of raster, especially relative to comparatively speedy gpu raster". And "gpu raster is limited to one worker thread at a time", because of locks on the graphics context, while image decodes can proceed in parallel.

Underneath most of this sits **Skia**, the 2D graphics library that its project page lists as the graphics engine for Chrome, ChromeOS, Android and Firefox, among others. Skia's GPU back end for years was **Ganesh**, built around OpenGL. In July 2025 Google announced that Chrome on Apple Silicon Macs had moved to **Graphite**, a new back end that runs on Dawn, Chrome's WebGPU implementation, as an abstraction over Metal, Vulkan and Direct3D. Google reported an improvement of "almost 15%" on the MotionMark 1.3 benchmark on a MacBook Pro M3. Other engines made their own bets. Firefox's **WebRender** moves most painting onto the GPU; Firefox 67 (21 May 2019) "gradually enabled" it "by default on Windows 10 desktops with NVIDIA graphics cards". **Vello**, an open-source Rust renderer, does 2D drawing with GPU compute shaders and describes itself as alpha software. Native Windows apps use Direct2D; Apple apps use Core Graphics.

FIGURE 4.1

Two raster paths, and the silent switch between them



The switch is a feature, not an error: no crash report, no alert, only a slower and costlier frame.

Chromium's own documentation: "a common reason for switching modes is that the gpu process has crashed too much and all of Chrome switches from gpu to software raster and compositing modes." Switching modes "destroys all resources".

The silent fallback

Engines do not trust every GPU. Chromium keeps a public list, `gpu/config/software_rendering_list.json`, of driver and hardware combinations on which it turns GPU features off, and a second list, `gpu_driver_bug_list.json`, of narrower workarounds. When a device matches, the engine quietly uses a slower path. The same happens at run time when the GPU process keeps crashing. Users see a slower, hotter app. Nobody sees an error.

Fallback does not only cost speed. It can also take features away. Some list entries switch off accelerated WebGL or WebGL2 outright (the measured counts below include them), and a software path does not have to offer every WebGL or WebGPU extension the GPU path offered. A screen that depends on one of those features loses it on that device, and the failure looks like a bug in the screen, not in the hardware.

The software paths are real products with real costs, and they are three different things. Chromium's own **software raster** is Skia running on the CPU, in the raster worker threads described above. Windows has **WARP**, Microsoft's software implementation of Direct3D, used when no hardware device is available, which Microsoft describes as a "high speed, fully conformant software rasterizer". Linux has Mesa's **llvmpipe**, "a software rasterizer that uses LLVM to do runtime code generation". **SwiftShader**, a CPU implementation of graphics APIs maintained by Google, is what Chromium has used as its software fallback for WebGL. On 13 February 2025 Chrome's team posted "Intent to Remove: SwiftShader

Fallback" on the blink-dev list, noting that the fallback served "approximately 2.7% of WebGL contexts". Under the plan, WebGL context creation fails instead of silently falling back, and the rollout has since been staged by platform. A kiosk application that draws its maps or charts with WebGL on a blocklisted GPU moves from slow to broken. To see which path a device took, open `chrome://gpu` in Chrome, or `edge://gpu` in Edge and in Microsoft's Edge WebView2 runtime (WebView2).

Layer	Windows	macOS	iOS	Linux	Android
Native 2D drawing	Direct2D	Core Graphics	Core Graphics	Cairo; Qt's own raster paint engine (Qt Widgets) or scene graph over its RHI (Qt Quick); Skia inside Chromium stacks, including Qt WebEngine	Skia, through HWUI
GPU API underneath	Direct3D 11 and 12	Metal	Metal	Vulkan, OpenGL	Vulkan, OpenGL ES
Chromium software rendering list entries for this OS (measured)	36	7	Not used: embedded web content is drawn by WebKit	21	11
System software rasterizer	WARP	Not covered here	Not covered here	llvmpipe (Mesa)	Not covered here

The measured row comes from Chromium's `software_rendering_list.json` on the main branch at commit `7e3211f5ea`, read on 18 November 2026: 82 entries in all, of which 30 turn off every GPU feature and 16 turn off GPU tile rasterization. By system: 36 Windows, 21 Linux, 11 Android, 7 macOS, 3 ChromeOS and 4 for any system. Windows has the most entries, so a Windows fleet manager should read the list first.

HiDPI: cost grows with the square of the scale

Display scaling multiplies raster work faster than it seems. Raster cost is roughly proportional to the number of pixels drawn, and a scale factor applies to both width and height. At 200 percent scaling, the same layout draws four times the physical pixels it drew at 100 percent; at 125 percent, about 1.56 times. A full-screen 1920 by 1080 frame is about 2.1 million pixels, or about 8.3 MB at four bytes per pixel. The same layout on a 3840 by 2160 panel at 200 percent is about 8.3 million pixels and 33 MB per full-screen buffer, before any extra tiles, layers or double buffering.

That arithmetic reaches budgets in three places. Raster time per frame grows, which matters most on low-power kiosk processors. GPU memory grows, because tiles, layers and decoded images are held at the higher resolution; when memory runs short, engines drop tiles and

re-raster them, or fall back. And remote display grows, because a VDI session at high resolution sends more pixels to encode. Windows adds one more effect: Microsoft's documentation explains that an app which is not DPI aware is rendered at 96 DPI and "bitmap-stretched", which "results in the application appearing blurry". Teams then fix the blur by declaring DPI awareness, and the raster bill arrives with the sharpness.

Decoders: the most reachable code on the path

Raster needs decoded images, so image decoders run on almost any picture a screen shows. They parse files from strangers. Two cases from September 2023 show the reach:

- **CVE-2023-4863.** A heap buffer overflow in **libwebp**, the WebP image library, fixed in Chrome 116.0.5845.187 and libwebp 1.3.2, reachable through "a crafted HTML page". CISA added it to KEV on 13 September 2023. Every product that bundled libwebp, including Electron and CEF apps, needed its own fix.
- **BLASTPASS.** On 7 September 2023 Citizen Lab reported a zero-click exploit chain that delivered NSO Group's Pegasus spyware to iPhones through PassKit attachments containing malicious images, sent over iMessage. Apple's fix for CVE-2023-41064 describes a buffer overflow where "processing a maliciously crafted image may lead to arbitrary code execution".

In Chromium, decoding happens in the sandboxed renderer, and the decoded pixels are uploaded to the GPU process. The sandbox limits the damage of a decoder bug; it does not remove the need to patch every copy.

HOW IT GOT THIS WAY

The arithmetic of combining images comes from Thomas Porter and Tom Duff's "Compositing Digital Images", published in the SIGGRAPH 1984 proceedings (Computer Graphics, volume 18, number 3, July 1984). Their operators, such as "source over", are still the names engines use. Skia began as Skia Inc., started by Michael Reed in Chapel Hill, North Carolina; Google acquired it in 2005, and in September 2008, as part of the Chrome release, open-sourced the graphics engine that was by then also the graphics core of Android. Early web engines rastered on the CPU. GPU raster came in steps: Ganesh in Chrome, WebRender in Firefox from 2019, Graphite in Chrome from July 2025. Each step moved more work to a GPU process that talks to a driver the engine vendor does not own, which is why the blocklist exists.

What hasn't changed: Porter and Duff's 1984 operators. Skia's current `include/core/SkBlendMode.h` still defines `kSrcOver`, their "source over", and groups the coefficient-based modes under the line `kLastCoeffMode = kScreen, //!< last porter duff blend mode` (Skia repository at `90f942bba5`, 17 November 2026). Skia uses "porter duff" there for every mode up to that marker; Screen itself is not one of Porter and Duff's 1984 operators. The file itself is young: commit `0591897548` created it on 3 October 2016 to "replace SkXfermode obj with SkBlendMode enum", and 21 commits have touched it since (measured 18 November 2026). The code was rewritten; the 1984 vocabulary stayed.

THE UNSEEN ANGLE

The most important rendering decision on Corvane's 3,000 kiosks may be made by a JSON file in another company's repository. Chromium's `software_rendering_list.json` decides, per GPU and driver, whether its engine trusts the hardware at all. It is public, it changes with releases, and an entry can move a whole kiosk model from GPU to CPU raster on the day a runtime update lands, with no crash and no log entry in the kiosk's own monitoring. Reading it before buying hardware, and checking `edge://gpu` (or `chrome://gpu` in a Chrome-based runtime) on each reference device after each runtime update, costs an hour. (Source: Chromium `gpu/config/software_rendering_list.json`, commit `7e3211f5ea`, 82 entries, read 18 November 2026; Chromium `docs/how_cc_works.md`.)

> *When the engine stops trusting your GPU, nothing crashes. Your fleet gets slower, hotter and more expensive, a WebGL screen can lose features or stop working, and no alert fires.*

This hop in the Frame Ledger

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
5 Paint: Blink paint to `PaintRecord`, Corvane check-in screen in its pinned WebView2 Fixed Version runtime (Chapter One)	Chromium project, largely Google	Corvane, when it moves the pinned runtime	Renderer process, sandboxed	n/a (commands, not pixels)	**PIN, RENT:** pinned with the runtime; re-test on each runtime move
6a Raster: `cc` TileManager and Skia GPU raster	Chromium and Skia projects, largely Google	Corvane, when it moves the pinned runtime; the mode can change at run time without a patch	GPU process (weaker sandbox, because it talks to the driver)	n/a until composited	**RENT, ISOLATE:** keep the GPU sandbox on; check `edge://gpu` per kiosk model after every update
6b Raster fallback: Skia CPU raster in the engine; SwiftShader for WebGL, while it remains a fallback; WARP, Windows' software Direct3D, underneath	Chromium and Skia projects for CPU raster and SwiftShader; Microsoft for WARP	Corvane, when it moves the pinned runtime; Microsoft, for WARP, through Windows Update. The switch to fallback is triggered by the blocklist or crashes, not by Corvane	Renderer or GPU process	n/a	**OWN** the policy: alert when a kiosk reports software raster; never ship WebGL screens without a non-WebGL path
6c Image decoders: libwebp, libpng and others inside the runtime	Open projects; one led by Google	Corvane, when it moves the pinned runtime; upstream fixes wait for that move	Renderer process, sandboxed	Page scripts, which can read decoded same-origin images through a canvas; every reader of the screen once composited	**RENT, ISOLATE:** load only images from Corvane's own servers on kiosks
6d GPU driver: one per kiosk model	GPU maker; Corvane pays in test time	GPU maker writes the fix; Corvane ships it in its image updates, quarterly or slower	User-mode and kernel-mode driver	n/a	**PIN:** tested versions only; check each against the blocklist before rollout

SECURITY LENS

Who can change it? The engine vendor, through its code and its blocklist; the GPU maker, through drivers; and anyone who can put an image or a WebGL scene on the screen, because decoders and shaders consume their input. The fleet owner changes it only by choosing hardware, drivers and runtime versions.

How fast can it be fixed? A decoder fix reaches an evergreen runtime in days and a pinned or bundled one when its owner ships. A driver fix depends on the GPU maker and on how often the fleet image is rebuilt, which for kiosks can be months. The GPU process is sandboxed more weakly than the renderer because it must talk to the driver, so its bugs and the driver's matter more.

What does AI change here? Local AI workloads now share the same GPU and the same video memory as the screen. A model running on a laptop or a VDI host competes with tiles and decoded images for memory, which can push the engine into dropping tiles or falling back. Treat GPU memory as a shared budget, and measure the screen while the AI feature is running, not alone.

BUSINESS IMPLICATIONS

Cost. Software raster turns GPU work into CPU work, which costs battery on laptops, heat and fan noise on kiosks, and host capacity on VDI. For Halbrook Bank's advisors on virtual desktops without a GPU, every frame is CPU raster, so screen design is a server cost: CPU raster on VDI means more processor cores per seat, and so larger or more hosts for the same number of advisors. For Corvane, a 4K kiosk at 200 percent scaling rasters four times the pixels of a 1080p one, and its hardware budget should say so.

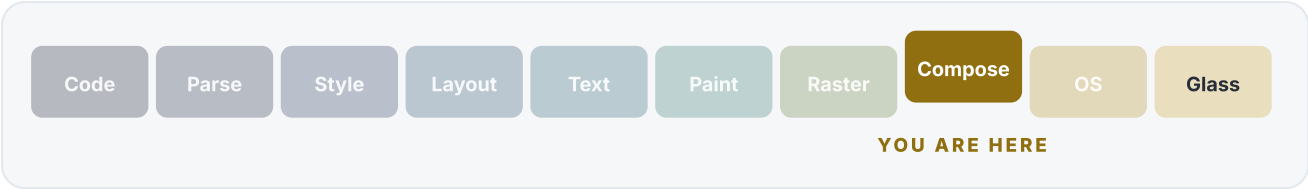
Visibility. The fallback is silent. Larkfield's crash reporting will never show it. A field metric for raster mode and GPU feature status, collected from the app per device model, is the only way to know how many customers are drawing on the CPU.

Value. Smooth scrolling, zoom and charts are what customers call quality, and they come from this hop. Qualifying devices by raster mode, not by "the app runs", lets Corvane buy hardware that stays fast for its whole life, and lets Larkfield tell customers which machines it supports with evidence.

Monday rule: on one reference device per model, record GPU raster on or off, the matching blocklist entries and peak GPU memory for your busiest screen; repeat after every runtime or driver update.

CHAPTER FIVE

Two Compositors, One Vsync



Larkfield Software has to decide where the video in its meeting view lives: drawn into the page with everything else, or handed to the hardware as its own layer. It sounds like a front-end detail. It decides whether the operating system's compositor can go to sleep, how much battery a laptop burns during a one-hour call, and which vendor you phone when a frame arrives late. This chapter is about the last two assemblers of the frame, and the handoff between them that nobody owns.

What a compositor does

A compositor takes finished pieces of a picture and assembles the final image. The pieces are called layers or surfaces: a scrolling list, a video, a toolbar, a window. Each piece has already been painted and rasterized (Chapter Four). The compositor decides where each one goes, which one sits on top, how transparent it is, and which parts actually changed since the last frame. That last job is called damage tracking, and it is how a compositor avoids redrawing a whole screen to blink a cursor.

A web or cross-platform app commonly has two compositors in the path. The first lives inside the engine. In Chromium it is called `cc`, the Chromium compositor. Chromium's own design document, `docs/how_cc_works.md`, says `cc` runs in both the browser process and the renderer process, and that the renderer uses a multi-threaded mode so scrolling and animation can continue while the page's main thread is busy. The document is candid that this mode "incurs a latency cost". The output of `cc` is a compositor frame. It is submitted to the display compositor, called `viz`, where a component named the surface aggregator combines the frames from all producers (the page, the browser's own interface, out-of-process frames) into one.

The other engine families make the same move in their own way. On Apple platforms, WebKit's web content process sends its layer tree to the app's process, where the code in `Source/WebKit/UIProcess/RemoteLayerTree` rebuilds it as platform layers for Core Animation, Apple's layer and animation framework. Firefox's WebRender, written in Rust,

takes over most of painting and compositing on the GPU; Mozilla began rolling it out in Firefox 67 in May 2019, to a subset of Windows 10 desktops with NVIDIA graphics. Each engine has its own compositor, its own threads and its own frame queue.

The compositor the OS owns

The second compositor belongs to the operating system. It takes the output of each window or app and builds the screen.

- **Windows:** the Desktop Window Manager, `dwm.exe`, introduced with Windows Vista. Microsoft's documentation for `DwmEnableComposition` says that as of Windows 8 "DWM can no longer be programmatically disabled". A request to turn it off returns success and does nothing. Apps reach it through DirectComposition and the Windows visual layer.
- **macOS:** WindowServer composes the desktop. Apps describe their content as Core Animation layers (`CALayer`), the same model iOS uses.
- **Linux:** on a Wayland desktop, the compositor is the display server. Mutter (GNOME), KWin (KDE), compositors built on the wlroots library, and Weston, the reference implementation, all speak the Wayland protocol. Apps hand it `wl_surface` objects.
- **Android:** SurfaceFlinger. Android's architecture documentation describes how WindowManager asks SurfaceFlinger for a layer whose core is a BufferQueue, a queue that passes graphics buffers between producer and consumer "by handle" rather than by copying. On each vsync, SurfaceFlinger walks its layers looking for new buffers, then asks the Hardware Composer (HWC) how to combine them.

That phrase, by handle, matters for cost. Compositing is not inherently a second copy of every pixel. Shared GPU surfaces and flip presentation let the OS compositor read the app's buffer where it already sits. Microsoft describes its flip model as a way to share content "directly with the desktop compositor, with minimal copies".

Question	Windows	macOS	iOS	Linux	Android
OS compositor	DWM (cannot be disabled since Windows 8)	WindowServer	Core Animation's render server, a separate system process (backboardd); apps do not address it directly	The Wayland compositor (Mutter, KWin, wlroots-based, Weston), or an X server plus a separate compositor	SurfaceFlinger with the Hardware Composer HAL
What the app hands over	Flip-model swap chain, DirectCompositio n visuals	Core Animation layers	Core Animation layers	`wl_surface` and subsurfaces	BufferQueue layers; `SurfaceContro l`
Path that skips full composition	DirectFlip, independent flip, multiplane overlays (MPO, since Windows 8.1)	Not documented in comparable detail by Apple; test on your hardware	Not documented in comparable detail by Apple; test on the device	Hardware planes exposed through kernel mode setting (KMS), used at the compositor's choice	HWC overlay planes; Android says devices "typically support four"
How you see which path ran	PresentMon; Windows Performance Analyzer	Xcode Instruments	Xcode Instruments on a connected device	Compositor debug output; varies by compositor	Perfetto; `dumpsys SurfaceFlinger`

Not always two: flips, overlays and bypass

The common picture of "engine compositor, then OS compositor" is the web-embed case, not a law. Native apps that draw one swap chain, hardware overlays and direct flips can take the OS compositor out of the per-frame path.

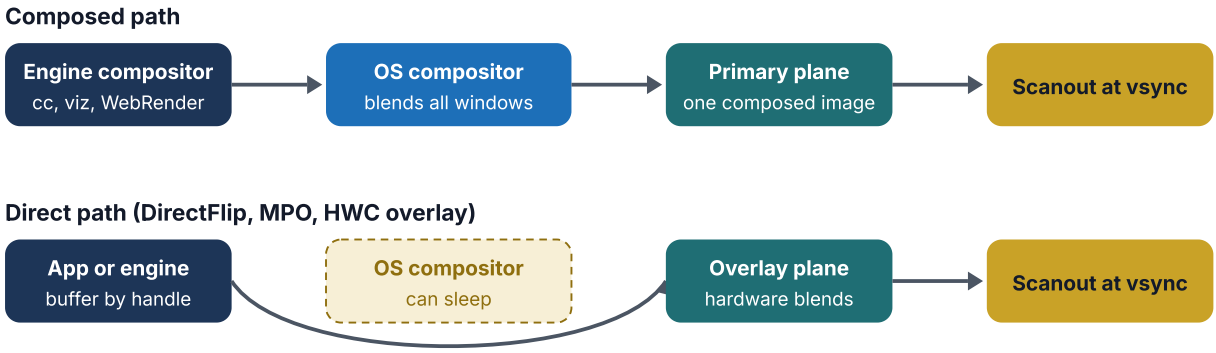
Microsoft's guidance "For best performance, use DXGI flip model" names three cases on Windows. In **DirectFlip**, a swap chain that matches the screen is shown directly instead of DWM's own. With **panel fitters**, the display hardware scales a slightly different-sized buffer on its way out. With a **multiplane overlay**, DWM reserves a dedicated hardware scanout plane for the app's buffer, so it can be "scanned out and potentially stretched to an alpha-blended sub-region of the screen". Once a swap chain is direct-flipped, the same page says, "the DWM can go to sleep, and only wake up when something changes outside of your application". Microsoft calls that state **independent flip**.

Android works the same idea from the other side. The HWC HAL marks each layer either for an overlay plane (device composition) or for the GPU (client composition), based on what the display hardware can do. Linux compositors can place a client buffer on a KMS plane when the formats and sizes allow. The direct path is a negotiation each frame. When another window, a notification or a cursor change gets in the way, the system may fall back to full composition. On Windows, Microsoft's page lists the other outcomes: DWM can "reverse compose" the content on top before flipping, or "leverage MPO to maintain the independent flip mode".

This is why Larkfield's video decision matters. A video promoted to its own plane can be scanned out without the GPU blending it into the page on each frame. Chromium has code for exactly this on Windows: `components/viz/service/display/dc_layer_overlay.h` keeps a list of the overlays "that are actually promoted" to DirectComposition. Whether promotion happens depends on the GPU, the driver, the video format and what sits on top of it. You cannot assume it. You can measure it.

FIGURE 5.1

One frame, two routes to the glass



The direct path is negotiated per frame and falls back to the composed path when anything covers the app.

The composed path is the common web-embed case. The direct path is what Microsoft calls DirectFlip and independent flip, and what Android's Hardware Composer does with overlay planes. Neither route requires a full pixel copy between the two compositors.

One vsync, two clocks

Vsync is the signal that the display is about to start drawing a new refresh. Both compositors schedule against it, but each runs its own clock and its own queue. The engine compositor decides when a page frame is ready. The OS compositor decides when the screen frame is ready. If the engine misses the OS compositor's deadline by a fraction of a millisecond, the frame waits a whole refresh.

Queue depth is the hidden variable. A flip-model swap chain needs at least two buffers, one being shown and one being drawn. Microsoft notes that in independent flip mode an app "can get down to 1 frame of latency on recent versions of Windows, with graceful fallback to the minimum possible when composed". Read that sentence the other way round: when the frame is composed, latency is higher, and the app does not control by how much. Add the engine's own pipelining and a busy machine, and a delay a user feels as "the app is slow" can live entirely in the handoff.

That is the visibility problem. A team that sees a dropped frame in the engine's trace blames its JavaScript. A team that sees it in Windows Performance Analyzer blames DWM. The truth often sits between the two queues, where neither vendor's tool shows the whole picture. Microsoft points developers to PresentMon, an open tool that reports which present mode actually ran: composed, DirectFlip or independent flip.

> *Two compositors do not double your pixels. They double your clocks, and the frame you lose lives between them.*

Idle composition and the battery

A screen that shows "nothing changing" can still cost power. Android's Hardware Composer documentation is unusually direct about it. It says overlay planes "may be less efficient than GL composition when nothing on the screen is changing", that caching a GPU-composited buffer "can improve the battery life of an idle device", and that "the number of layers used by an app can have a measurable impact on power consumption and performance".

The lesson is not that overlays are good or bad. It is that the right answer depends on whether the content is moving. A playing video benefits from its own plane. A static check-in screen may be cheaper as one flattened buffer. Microsoft makes a similar point for Windows: the flip model gives "lower power usage", and in independent flip the DWM sleeps. An app that keeps animating a spinner it hid behind another element keeps both compositors awake.

For Larkfield, the meeting view is the expensive case: a moving video, a moving self-view, chat updates and a toolbar on top. For Corvane, the expensive case is the opposite: a kiosk attract loop or a signage page whose "idle" animation means the GPU and compositor never rest for months. Neither cost shows up in a design review. Both show up on the power bill and in device heat.

The compositor as a security boundary

Because the OS compositor holds the pixels of all windows, its design decides who else can read yours. Under X11, the old Linux display protocol, the answer is "most clients". Chromium's security FAQ states it plainly: "The X11 protocol does not isolate clients from each other." It treats a compromised GPU process attacking the browser over X11 as outside its threat model for that reason.

Wayland was designed the other way. The core protocol gives a client its own surfaces and no request to read another client's window. Screen capture on GNOME and KDE goes through the desktop portal's ScreenCast interface, where calling `Start()` "typically

presents a dialog to the user" before any stream is handed out. GNOME 49, released in September 2025, disabled the GNOME X11 session; X11 apps keep running through XWayland. That bridge carries the old weakness with it. XWayland is one X server that runs as a single Wayland client for all the X11 apps on the desktop, so those apps can still see each other's windows and input, though not those of native Wayland apps. A signage app moved to Wayland is only isolated once it runs as a native Wayland client. For Corvane's Linux signage, the migration is a security decision as much as a graphics one: what can a compromised signage app see on a Wayland screen, compared with on an X11 one?

HOW IT GOT THIS WAY

Composited desktops arrived on personal computers in the 2000s. Apple's Mac OS X reached stores on 24 March 2001 with its Quartz graphics layer and the Aqua interface. Windows Vista (2007) brought the Desktop Window Manager, which renders each window to an off-screen buffer and composes the desktop from them. Windows 7 added the first flip-style presentation (Direct3D 9's FLIPEX), Windows 8 brought the flip model to DXGI and made DWM permanent, and Windows 8.1 (WDDM 1.3) added multiplane overlays. On Linux, Kristian Høgsberg started Wayland as a side project in 2008; Wayland and Weston 1.0 were announced on 22 October 2012. The X11 era is now ending on the largest desktop: GNOME 49 (September 2025) disabled its X11 session.

Today's decision follows from that history. On Windows, you cannot opt out of DWM, so you design for its fast paths. On Linux, the signage fleet's move to Wayland changes who can capture the screen.

What hasn't changed: the flip at vsync between at least two buffers. Microsoft's flip-model requirements still open with "The buffer count must be at least 2" (Microsoft Learn, "For best performance, use DXGI flip model", checked 17 November 2026). On Linux the repository history shows the same continuity: Weston, the Wayland reference compositor, still carries Kristian Høgsberg's "Initial commit." of 30 September 2008 (commit `97f1ebe8`) as the first of 11,544 commits (Weston repository at `749e4077`, measured 17 November 2026).

THE UNSEEN ANGLE

The idle cost of a screen is decided in the handoff, and the vendors document it in places a CTO rarely reads. Android's HWC page says a static screen can be cheaper as one cached GPU buffer than as overlays; Microsoft's flip-model page says the DWM sleeps only while a swap chain stays direct-flipped. Neither the engine vendor nor the OS vendor measures your screen's idle power for you. A team that wants a battery or heat number for a kiosk or a meeting view has to measure the composition mode itself, with PresentMon on Windows or Perfetto on Android. Sources: source.android.com, "Hardware Composer HAL"; Microsoft Learn, "For best performance, use DXGI flip model".

This hop in the Frame Ledger

Larkfield's meeting view on a Windows laptop, with the bundled engine. Hop 7, the engine compositor, has two sub-rows here: 7a for `cc`, which builds compositor frames, and 7b for `viz`, which aggregates them and decides overlay promotion.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
7a Engine compositor: Chromium `cc`	Chromium upstream; Larkfield pays to ship it inside its app	Larkfield, at its own engine-update pace	Renderer (sandboxed) and browser process	Nobody outside the engine	**RENT, OWN:** rent the code; own the update cadence
7b Display compositor: Chromium `viz`, including video overlay promotion	Chromium upstream; Larkfield ships it	Larkfield, with the engine	GPU process	Nobody outside the engine	**RENT:** measure overlay promotion on the top five GPU models in the customer base
8 OS compositor: DWM	Microsoft; paid through the Windows license	Microsoft, through Windows servicing	`dwm.exe`; a windowed frame usually passes through it, except where independent flip or MPO scanout takes it past DWM	The user, capture APIs, screen-reading tools	**RENT:** design for independent flip where the view allows it
9 Scanout: multiplane overlay (MPO) plane for the promoted video	GPU vendor and driver	GPU vendor, through driver releases	Kernel-mode display driver	Same as the composed screen, for capture purposes	**RENT:** record the driver versions where promotion works

The same meeting view on Larkfield's Mac build swaps hop 8 for WindowServer, owned and patched by Apple through macOS updates and readable by the user and by apps the user grants screen recording. The decision there is also RENT, with one test added: idle power on the reference Mac.

SECURITY LENS

Who can change it? The engine compositor belongs to whoever ships the engine: Larkfield for its bundled copy, Microsoft for Halbrook's PCs, which run Microsoft's Edge WebView2 runtime (WebView2), the OS vendor for a system web view. The OS compositor belongs to the OS vendor or, on Linux, to the desktop project the distribution ships. The GPU vendor controls whether overlays work at all.

How fast can it be fixed? A compositor fix reaches users as fast as the slowest of three channels: your engine release, the OS update and the GPU driver. A Linux signage fleet on a long-term distribution may wait for a point release or a backport.

What does AI change here? Screen-reading agents take one of two routes: they read a capture of the OS compositor's output, or they read the accessibility tree and never see the pixels at all, as the prologue showed with UI Automation. When overlays, protected surfaces or a cursor plane are involved, what the agent captures can differ from what the panel showed (Chapter Nine). **Our view (as of November 2026):** as agents move onto regulated desktops, the composition mode of a screen becomes part of its audit evidence, not only its performance profile.

BUSINESS IMPLICATIONS

Cost. Larkfield pays for composition in battery and support calls ("my laptop gets hot in meetings"). Corvane pays for it in power and heat across 3,000 kiosks and its signage. Halbrook pays on VDI, where there is no local overlay plane and, because its VDI hosts have no GPU (Chapter Four), composition runs on the host CPU along with raster. Price each case by measuring, not by assuming the fast path.

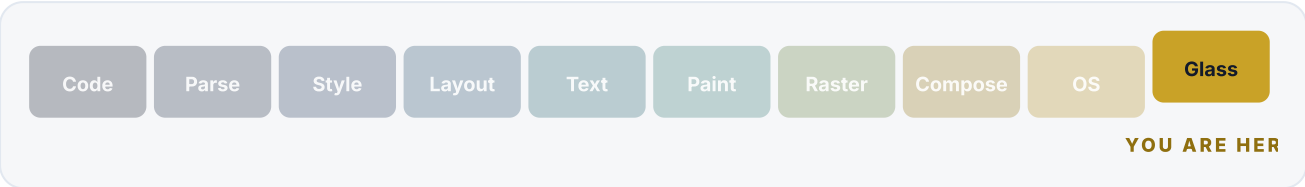
Visibility. Engine traces and OS traces each show half the handoff. Add PresentMon (Windows) and Perfetto (Android) to the performance runbook, and record the present mode next to frame time for each critical screen.

Value. A video that stays on its own plane and a static screen that lets the compositor rest are product features: longer battery, quieter fans, cooler kiosks. On Linux signage, the move to Wayland also narrows who can read the screen.

Monday rule: for your heaviest screen, capture one minute of presentation data on a reference device, write down which present mode ran, and decide whether video and static content should be composed or promoted.

CHAPTER SIX

The Glass: Display, Color, HDR and Timing



Corvane Hotels is about to approve the next panel model for its check-in kiosks, and the purchase order lists size, brightness and price. It does not list what the panel does to color, how it behaves when content protection fails, how it ages under a logo shown twenty hours a day, or what the kiosk does when the GPU resets. Those are the parts of the frame that happen after your software is finished with it. This chapter is about the last hop, the glass, and why it belongs in your requirements.

Scanout: the frame leaves the computer

When the OS compositor (Chapter Five) finishes a frame, it does not "send a picture". The GPU's display engine reads the finished buffer and streams it down the cable, line by line, at the display's refresh rate. This is called scanout. The panel's own controller receives the stream, may scale it, may process its color, and drives the pixels.

The refresh rate sets the deadline. At a fixed 60 Hz, a new refresh starts every 16.7 ms, and a frame that misses one waits for the next. At 120 Hz the interval is 8.3 ms. With variable refresh, the display waits for the frame instead of the frame waiting for the display, within a range the panel supports. VESA added Adaptive-Sync to the DisplayPort 1.2a standard on 12 May 2014, bringing to external monitors a technique that had been part of embedded DisplayPort since 2009. Apple introduced ProMotion with the iPad Pro models of June 2017, at up to 120 Hz and lower rates when content is still.

For a kiosk, the practical point is that the deadline is a property of the panel and its settings, not of your code. A screen that is smooth on the designer's 120 Hz laptop may stutter at 60 Hz on the kiosk, and a variable-refresh panel can hide a slow frame that a fixed panel exposes.

Color: a number becomes a color somewhere else

An app writes numbers. The color the eye sees depends on how those numbers are interpreted on the way to the glass. For decades the default interpretation has been sRGB, a color space created by HP and Microsoft in 1996 and standardized as IEC 61966-2-1, published on 18 October 1999. sRGB describes the monitors of its era. Modern panels can show more saturated colors (a "wide gamut"), so the same number can look different on two screens unless something converts it.

Who does that conversion differs by platform, and on Windows it depends on a mode. Microsoft's documentation is precise: when Advanced Color is inactive, "Windows doesn't perform any color management on your app's visual content output" and "assumes that your app content is in the standard sRGB color space". When Advanced Color is active, Windows performs automatic system color management, converting to the display's color space "determined by the current default color profile. If there is no profile, then EDID colorimetry data will be used." EDID is the block of data a monitor reports about itself. Two names are easy to conflate here. **Advanced Color** is Microsoft's umbrella term for HDR, wide color and high bit depth; its HDR guide says Advanced Color "first shipped" for HDR displays with Windows 10 version 1709, with system color management for those HDR displays. **Auto Color Management (ACM)** is the later extension of that color management to SDR displays: Windows 11 version 22H2, only on SDR displays "specially provisioned" with a valid profile, and with a WDDM 3.0 driver. On an HDR display the two stages are split: the DWM converts each app into its own FP16 composition space before blending (the OS compositor, hop 8), and the display kernel then converts the framebuffer to the panel's signal (the display, hop 9), clipping colors the panel cannot show.

Android lets an activity opt in to wide color with `android:colorMode="wideColorGamut"`, from Android 8.0. The developer guide adds two warnings worth reading twice. The system will not grant the mode if the display is wide-gamut capable but not color-managed. And when the mode is on, "the activity's window uses more memory and GPU processing for screen composition". On Linux, a common color-management protocol for Wayland, `color-management-v1`, entered wayland-protocols in release 1.41 on 17 February 2025, and GNOME 48's Mutter added support. Before that, color management on the Linux desktop was largely the business of each app.

Question	Windows	macOS	iOS	Linux	Android
Who converts app color to the panel	The app, unless Advanced Color is active; then Windows (ACM)	ColorSync, with a color profile per device	The system; apps tag the color space of their content	The app, unless the compositor supports `color-management-v1` (wayland-protocols 1.41, 2025)	The system, per activity, when the app opts in to wide color (Android 8.0)
Where the panel's description comes from	The default color profile, else EDID colorimetry	The display's profile, which ColorSync Utility can change	Not exposed to apps	The profile the compositor loads	Not exposed; the system decides whether to grant wide color
HDR and wide color for apps	HDR swap chain formats; content not clamped unless composed to an SDR desktop	Extended dynamic range (EDR), with headroom that depends on the display	EDR with a "headroom" (`UIScreen.currentEDRHeadroom`) that changes with screen state	Through `color-management-v1` , compositor and driver dependent	Wide color costs "more memory and GPU processing for screen composition"
What a kiosk team must pin	Advanced Color and HDR settings, driver version	Display mode, color profile and brightness policy	Device model, iOS version and auto-brightness setting	Compositor version and profile	Device model and firmware

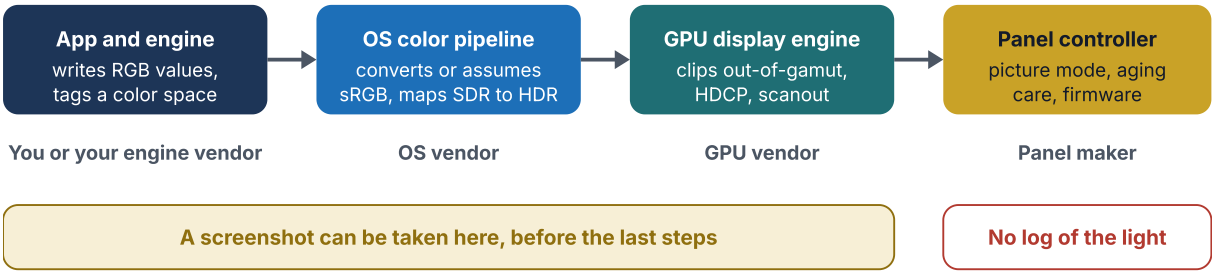
HDR: the meaning of a pixel value changed

High dynamic range (HDR) changes more than brightness. ITU-R Recommendation BT.2100, first approved in July 2016, defines HDR signals with two transfer functions, PQ and HLG. A PQ value names an absolute brightness. An SDR value names a fraction of whatever the display's white happens to be. So the same app can be shown three ways: on an SDR desktop, on an HDR desktop with its SDR content mapped up, and as true HDR content.

The platforms say so in their own words. Microsoft notes that HDR content in flip-model swap chains "isn't clamped unless it's composed to an SDR desktop", which means a composition decision (Chapter Five) can change the brightness range of your pixels. Apple's documentation for `UIScreen.currentEDRHeadroom` says the screen's current headroom limits all rendered content and can change with the screen's configuration. A brand color or a red warning is therefore not one value. It is a value transformed by the app, the engine, the OS compositor, the GPU's display pipeline and the panel's controller before it reaches the eye.

FIGURE 6.1

Four owners between a color value and the eye



Only the first owner works for you. The others decide what your color becomes.

The color path on a modern desktop or kiosk. Windows converts color in the OS pipeline only when Advanced Color is active; otherwise it assumes sRGB. The GPU's display pipeline clips out-of-gamut colors. The panel applies its own picture mode. Nothing in the chain records what was emitted.

Color after capture

Because conversion happens late, a screenshot is usually a copy of a composed surface taken before the last steps. On Windows with Advanced Color active, the conversion to the panel's color space and the GPU's clipping of out-of-gamut colors come after the app has rendered. The Desktop Duplication API, used by many recording and remote-support tools, protects protected video content from capture and can deliver the mouse pointer separately from the frame, so the capturing tool must draw it back in. The format the tool asks for matters too: Microsoft's screen-capture guide recommends a 16-bit floating-point format for HDR content, to avoid "pixel overclipping (i.e. the captured content looks washed out)" (the same quote in Chapter Nine). A screenshot can therefore differ from what the panel showed because of overlays, protected content, the cursor plane, and HDR or color processing after capture.

For most screens this does not matter. For a few it does: a medical image, a safety warning whose meaning is its color, a brand color under contract, or a regulated screen where someone may later ask "what did the customer see?". Treat color on those screens as business logic. State the color space you render in, the displays you approve, and the test that proves a critical color survives the chain. A design sign-off made on a calibrated monitor is not evidence of what ships to a wide-gamut kiosk.

HDCP: the screen that goes dark on purpose

High-bandwidth Digital Content Protection (HDCP) encrypts the link between a source and a display so protected video cannot be copied off the cable. Its specification, revision 1.0, is dated 17 February 2000 and carries Intel's copyright. When the devices on the link cannot authenticate each other, protected content is not sent. What a person sees then depends on the source device: often a black window or a blank screen. Intel's own support article for one such case describes a second display that "goes out and does not function" during HDCP playback, fixed by a newer graphics driver.

For Corvane, HDCP matters wherever a kiosk or signage player plays licensed video through an adapter, a splitter or a long cable run. The failure sits between the GPU driver and the panel, below the app. Your monitoring sees a healthy app. The guest sees a dark screen.

Burn-in and GPU resets: the glass as availability

Burn-in. OLED panels age pixel by pixel, so content that never moves wears a ghost into the panel. RTINGS' long-term television tests found that long periods of static content cause burn-in on OLED TVs. Panel makers ship mitigations, such as LG's screen shift and logo luminance adjustment, inside panel firmware you usually cannot patch yourself and often cannot audit. A signage page with a fixed logo and clock, shown for months, is the worst case. Rotate layouts, move static elements, and buy panels rated for commercial duty cycles.

GPU resets. Windows detects a GPU that stops responding through Timeout Detection and Recovery (TDR). Microsoft's driver documentation gives a default timeout of two seconds, after which the OS resets the GPU and purges video memory. The visible artifact is "a screen flicker" and the message "Display driver stopped responding and has recovered". It also warns that "some legacy DirectX applications might just render black at the end of this recovery", and that an application "must release and then re-create its Microsoft Direct3D device". Repeated hangs stop the machine: by default, in Microsoft's words, "the OS bug-checks the computer on the sixth (or more) GPU hang when it detects that five (5) or more GPU hangs (0x117) and subsequent recoveries occur within one (1) minute." For a kiosk, a GPU reset is an availability event. The question is not whether it happens but whether the kiosk app recovers by itself or shows black until a technician arrives.

> *The panel is the hop with the weakest update channel, no log and no owner on your org chart. It is also the only one the guest looks at.*

HOW IT GOT THIS WAY

sRGB was created by HP and Microsoft in 1996 and standardized by the IEC in 1999, for the cathode-ray tubes of the day. It became the assumed color of untagged content, which is why so much software still does no color management at all. HDCP followed in 2000, adding content protection to digital display links. Variable refresh reached external monitors through VESA Adaptive-Sync in May 2014, and Apple's ProMotion brought adaptive refresh to tablets in 2017. ITU-R BT.2100 (July 2016) gave HDR values a new meaning. Operating systems then had to manage color for apps that never asked: Windows 10 version 1709 for HDR displays, Windows 11 version 22H2 for SDR displays, and on Linux a common Wayland protocol in February 2025.

Today's decision follows from that sequence. Color management moved from the app to the OS at different times on different platforms, so the same screen can be color-managed on one kiosk model and not on another.

What hasn't changed: the sRGB assumption. Microsoft's documentation still says that for apps that are not Advanced Color-aware and render to a standard integer format, "Windows will explicitly treat the app as sRGB" (Microsoft Learn, "ICC profile behavior with Advanced Color", checked 17 November 2026).

THE UNSEEN ANGLE

The monitor's self-description can become your color truth. When Advanced Color is active and no color profile is installed, Windows uses the display's EDID colorimetry data, the numbers the panel reports about itself, to convert your app's colors. EDID comes from the panel maker's firmware. It is not signed and not updated through any channel you manage, and nothing in the platform checks that its numbers match the individual panel in front of the guest. A kiosk fleet that relies on automatic color management without installing approved profiles is trusting each panel's own claim about what red is. Source: Microsoft Learn, "ICC profile behavior with Advanced Color".

This hop in the Frame Ledger

Corvane's self check-in kiosk, Windows IoT, a web check-in screen, and an approved panel model:

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
9 Scanout and display: refresh (GPU display engine, panel timing)	GPU vendor and panel maker; Corvane pays in hardware	GPU vendor by driver release; panel maker rarely	Kernel-mode display driver	Nobody logs it	**PIN**: the driver and panel model; test at the panel's real refresh rate
8 OS compositor: DWM color conversion (sRGB default; FP16 composition space when Advanced Color is active)	Microsoft	Microsoft, through Windows servicing	DWM, user mode	Capture tools see this composed frame; an 8-bit capture clips HDR, a 16-bit float capture keeps its range	**OWN**: the policy; set the color mode (HDR, ACM or off) by image
9 Scanout and display: framebuffer to panel color (display kernel, driver, profile or EDID)	Microsoft, GPU vendor, panel maker	Windows servicing and driver releases	Display kernel and driver	Nobody; capture happens before this conversion	**PIN**: driver version and approved display profiles
9 Scanout and display: HDCP link (GPU, adapter, cable, panel)	GPU vendor, adapter maker, panel maker	Driver updates; adapters and panels rarely	Driver and display hardware	Nobody; the app is not involved	**PIN**: approved adapters; test licensed video on the full chain
10 Glass and readers: OLED or LCD panel, aging care in firmware	Panel maker; Corvane pays in replacements	Panel maker, by firmware; often no reliable field update channel	Not applicable	Guests, staff, any camera	**OWN**: the content rules; no permanent static elements; commercial-rated panels
9 Scanout and display: GPU reset (TDR) recovery	Microsoft and GPU vendor; Corvane pays in truck rolls	Corvane, in its own kiosk app	Kiosk app and engine GPU process	Event log	**OWN**: recovery; recreate devices, watchdog, alert on repeated TDRs

SECURITY LENS

Who can change it? The OS vendor controls the color pipeline, the GPU vendor controls scanout and HDCP behavior, and the panel maker controls the controller firmware, its picture modes and its aging care. Anyone with physical access can change a panel's picture settings or swap a cable.

How fast can it be fixed? Driver issues are fixed at the GPU vendor's pace, then at yours once you test and roll out. Panel firmware often has no reliable field update channel, so its issues may never be fixed in the field, which makes the approved-panel list the main control.

What does AI change here? Agents that judge a screen from a capture judge the frame before color conversion and panel processing. Our view (as of November 2026): for screens where color carries meaning, teams will need a stated color assumption and a physical test, because neither a human screenshot nor an agent's capture shows the light that left the glass.

BUSINESS IMPLICATIONS

Cost. Corvane's glass costs are panel replacements from burn-in, technician visits for dark screens after a GPU reset or an HDCP failure, and power for high-refresh or HDR modes nobody needed on a check-in screen. Larkfield pays in support tickets when the same color looks different on a customer's wide-gamut laptop. Halbrook pays when a branch's customer-facing screen fails without an app error.

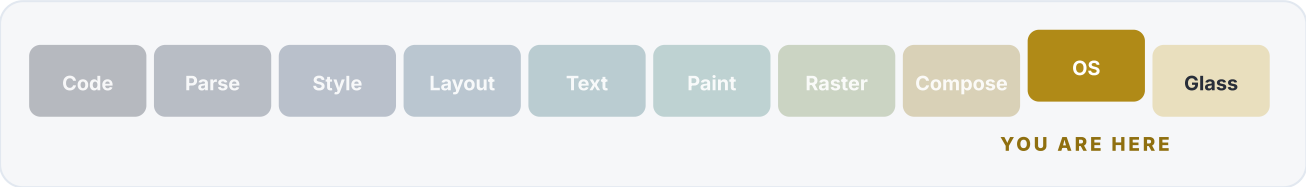
Visibility. No standard tool logs what a panel displayed. Collect what you can: TDR events from the Windows event log, display mode and color settings by device, panel model and firmware in the asset register.

Value. A short approved-display list, tested color for the few screens where it matters, and a kiosk app that recovers by itself after a GPU reset turn the glass from an unknown into a managed hop.

Monday rule: add three lines to your next display purchase: the color mode it will run in, the commercial duty cycle it is rated for, and the test your app must pass after a forced GPU reset on that model.

CHAPTER SEVEN

Processes, Sandboxes and the GPU as the New Kernel



Halbrook Bank's teller desktop embeds Microsoft's Edge WebView2 runtime, and somewhere in a build script or a branch PC's environment there may be a line that turns its sandbox off. Nobody on the risk register would choose that. It happens because a flag fixed a crash once, or a support engineer set a variable to debug a printer. This chapter shows how the engine splits its work into processes, what each sandbox actually blocks, where the GPU process sits in that picture, and which switches an embedding app can flip without anyone noticing.

Why a browser became many processes

Google launched Chrome on 2 September 2008. Nine days later, the Chromium blog explained its design: a browser process that owns the windows, tabs and user interface; renderer processes that parse and draw web content; and plug-in processes started on first use. Chromium's own history of its process models says the launch default was process-per-site-instance: a new renderer for a new site typed in the address bar, or for another instance of a site in another tab. Link clicks and cross-site frames stayed in the same process, so in practice it was close to one renderer per tab. One misbehaving tab no longer took down the others. The same team was open about the cost. A later Chromium post on memory acknowledged that the multi-process design uses noticeably more memory than a single-process browser when many tabs are open, and argued the trade was worth it.

That split is the foundation of everything in this part of the book. The renderer is the process that touches untrusted input: HTML, CSS, JavaScript, images and fonts from the network. If it is compromised, the question becomes what that process can reach. The answer is set by the sandbox, and by every other process the renderer is allowed to talk to.

What the renderer sandbox blocks

A sandbox is a set of operating system restrictions applied to a process so that, even running attacker-controlled code, it cannot open files, sockets or devices on its own. It must ask a more trusted process, the browser, over inter-process communication (IPC).

- **Windows.** Chromium's sandbox design document describes the renderer running with a restricted token that has no privileges and the lowest Untrusted integrity level, so "almost all resources that the renderer process uses have been acquired by the Browser". The same document gives the integrity ladder, which it dates itself ("At M75", that is Chrome 75) and says "will change over time": the browser process at Medium, the GPU process at Low, and most other services, including isolated renderers, at Untrusted. Each renderer runs in its own job object. Renderer processes also set `DisallowWin32kSystemCalls`, which, in the document's words, "significantly reduces the kernel attack surface available from a renderer".
- **Linux.** Chromium uses two layers. Layer one, built on unprivileged user namespaces where the kernel supports them, separates processes and can cut network access. Layer two installs a seccomp-bpf program that the kernel checks on each system call, to "restrict access from a process to the attack surface of the kernel".
- **macOS.** Chromium applies Seatbelt, the sandbox(7) mechanism, with policies that start from `(deny default)` and are named after the process type they apply to.
- **Android.** From Android 8.0, the system WebView runs web content in a separate sandboxed renderer process on most devices; low-memory devices may fall back to a single process. An app can survive a renderer crash by overriding `onRenderProcessGone()`.

Question	Windows	macOS	iOS	Linux	Android
Sandbox primitive used by the engine	Restricted token, Untrusted integrity, job object, win32k lockdown	Seatbelt (sandbox(7)) profiles, default deny	Set by Apple; apps embed WebKit only	User namespaces plus seccomp-bpf	Sandboxed renderer process (Android 8.0 and later on most devices)
Embedded runtime process model	WebView2: one browser process per user data folder, plus renderers, GPU and helpers	WKWebView: WebKit's out-of-process model, with web content, networking and GPU processes	WKWebView: the same WebKit process model, run by the system	Depends on the runtime: CEF, Qt WebEngine and Electron follow Chromium; WebKitGTK follows WebKit	System WebView: renderer process outside the app
Site Isolation in the embedded runtime	Yes in WebView2; can be turned off by a flag	WebKit's own process policy	WebKit's own process policy, set by Apple	Follows the embedded engine	Not supported: "Android WebView does not yet support multiple renderer processes or out-of-process iframes" (Chromium process model document, checked 18 November 2026)
What the embedder can switch off	Browser flags in code or an environment variable	Little for WKWebView; a bundled Chromium (Electron, CEF) takes Chromium switches	Little; Apple sets the policy, and other engines need an EU or Japan entitlement	Engine flags, packaging options	Little; the system WebView sets the policy

Site Isolation and its memory bill

The original model kept sites apart most of the time, not all of the time. In January 2018, the Spectre class of attacks, then made public, showed that code in a process could read memory in that same process through speculative execution. If two sites shared a renderer, one could read the other. Chromium's answer was Site Isolation: no two sites share a renderer process, including cross-site frames inside a page. That is not one process per site. Chromium's process model document groups documents into site instances, and it reuses same-site processes: on desktop it sets a "soft" process limit based on memory and, above it, starts "randomly reusing same-site processes".

Chromium's Site Isolation page states that it "was enabled by default for all sites in Chrome 67 on Windows, Mac, Linux, and Chrome OS", and that on desktop "this is about 10-13% when isolating all sites with many tabs open". That is a many-sites figure: an app that loads one origin with few cross-site frames creates fewer extra processes and pays less. On Android, Chrome 77 isolated only the sites users log into, on devices with at least 2 GB of RAM, at "about 3-5% overhead". The same page notes that Site Isolation "is not yet

supported in Android WebView", and Chromium's current WebView security notes say there is "generally only one renderer process per browser context", so there is "little process-level site isolation in WebView today". Mozilla shipped its equivalent, Project Fission, to desktop Firefox in version 95 in December 2021.

For an embedding app, the bill multiplies. Microsoft's process model page for WebView2 says a process group has "a single browser process", one or more renderers, and helpers "such as the GPU process". All of them are tied to one user data folder, and an app that uses several user data folders gets a separate process collection for each. On a shared VDI host, ten apps that each embed the runtime are ten process groups, each with its own GPU process and renderers.

The GPU process and the command buffer

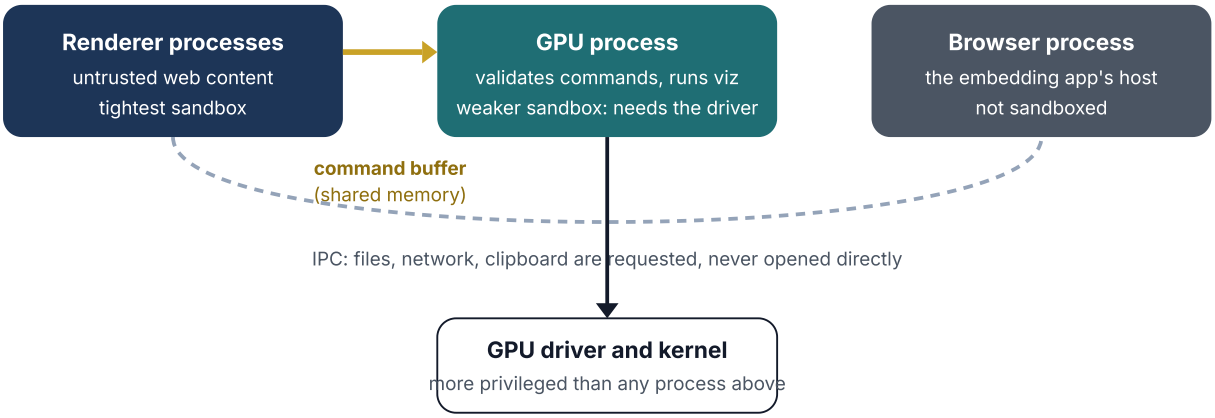
The renderer is sandboxed so tightly that it cannot talk to the graphics driver. Yet it needs the GPU to draw. In August 2010, the Chromium blog described the fix as "a new process (yes, another one) called the GPU process", which "accepts graphics commands from the renderer process and pushes them to OpenGL or Direct3D (via ANGLE)". ANGLE is the library that translates one graphics API into another.

The channel between them is the command buffer. Chromium's design document describes it: the renderer "writes commands into some shared memory" and tells the GPU process over IPC how far it has written. The GPU process then, "for each command", validates "the command, its arguments, and whether or not the arguments are appropriate" before calling into the OS. The document gives the design rule in five words: "Assume the client can go rogue." Its stated goal is blunt too: "Graphics systems in OSes have gaping security holes," and the first job of the GPU process is to keep web content from reaching them.

WebKit made the same move later. The commit "Introduce a GPU process" (`2d9d0c575ad`) landed on 4 December 2019, and its source tree now carries a `Source/WebKit/GPUProcess` directory with its own entry point and media, WebRTC and graphics code, so canvas, media and WebGL work can leave the web content process.

FIGURE 7.1

Who talks to whom, and how tightly each is held



Chromium on Windows (M75 and later): renderers at Untrusted integrity, GPU process at Low, browser at Medium.

The GPU process is a shared service between many sandboxed renderers and the most privileged code on the device. That position, not its privilege, is why it draws attackers' attention.

"The new kernel": a metaphor with limits

People sometimes call the GPU process "the new kernel". The metaphor holds in one sense and fails in another, so use it carefully.

It holds because the GPU process plays a kernel-like role for the engine. It is shared by renderers from many sites. It accepts a stream of requests through a narrow interface, validates them, and forwards them to hardware. A flaw in that validation, or a driver bug the validation does not catch, is a route out of a renderer sandbox, the way a system call bug is a route out of a user process.

It fails because the GPU process is not the most privileged code on the machine, and it is sandboxed. Chromium's own document places it at Low integrity on Windows, above the renderer's Untrusted level and below the browser process at Medium. Its sandbox is weaker than the renderer's because it must talk to the graphics driver, which runs with far more privilege than the GPU process does. Platform details also matter: Chromium's security FAQ says a compromised GPU process attacking the browser over X11 is not treated as a security bug, because "the X11 protocol does not isolate clients from each other". The honest form of the claim is this: the GPU process is a weaker sandbox sitting next to the driver, and it is shared by all the content you render.

> *A sandbox is a promise made by the engine vendor and kept, or broken, by whoever writes the launch flags.*

What an embedding app can switch off

An app that embeds an engine inherits its process model and its sandboxes. It can also take them away. Microsoft's WebView2 browser-flags page lists flags that include `no-sandbox` ("Disables the sandbox for all process types that are normally sandboxed"), `disable-site-isolation-trials` ("Disables site isolation") and `disable-gpu`. Chromium's own `disable-gpu-sandbox` switch is not on Microsoft's list, which is one more reason to block every unlisted switch rather than trust the list. Flags can be passed in code through `AdditionalBrowserArguments`, or set on the device through the `WEBVIEW2_ADDITIONAL_BROWSER_ARGUMENTS` environment variable or registry keys. Microsoft is explicit: "Apps in production shouldn't use WebView2 browser flags", and if you set them in code, "be sure to remove the flags in code before shipping your app". It also notes that elevated apps ignore flags set through the device environment.

Why would a team turn a sandbox off? The usual reasons are practical: a legacy native module or plug-in that needs file or device access from the renderer, a crash on one GPU driver that a flag made go away, or a debugging session whose flags were never removed. Each is a real problem with a better fix than removing the sandbox.

Electron changed its default in version 20 (2022): renderers are sandboxed "unless `nodeIntegration: true` or `sandbox: false` is specified". An Electron-style app built before that, or one that turned the sandbox off to load a native module in the renderer, may still run unsandboxed today. CEF and Qt WebEngine pass Chromium switches through their own configuration. Electron, WebView2 and Qt WebEngine ship the sandbox on. CEF supports it but needs the embedder to integrate it: on Windows from M138 the app is built as a DLL and launched through CEF's `bootstrap.exe` (earlier versions linked the `cef_sandbox` library), and on macOS each helper process loads `libcef_sandbox.dylib` and calls `cef_sandbox_initialize()`; passing a null sandbox pointer simply runs without it. In every case a single line in the embedder's code or packaging can leave the sandbox off, so each deployment must verify that it is on in the processes it actually runs.

HOW IT GOT THIS WAY

Chrome shipped its multi-process design in September 2008. The GPU process followed in 2010, when hardware-accelerated compositing needed a way for sandboxed renderers to use the GPU. Spectre, made public in January 2018, pushed Chrome to isolate all sites on desktop in Chrome 67 (2018) and logged-in sites on Android in Chrome 77 (2019). Mozilla shipped Fission in Firefox 95 (December 2021). Electron made the renderer sandbox the default in version 20 (2022). Each step tightened the default and added memory cost, and each left a switch to undo it.

Today's decision follows from that sequence. The defaults are now strong. The risk has moved to the embedder's configuration and to the processes that must stay less sandboxed, such as the GPU process.

What hasn't changed: the renderer still asks the browser for what it needs. The 2008 design of a renderer that acquires nothing itself is the same rule Chromium's sandbox document states today, that "almost all resources that the renderer process uses have been acquired by the Browser" (`docs/design/sandbox.md` , Chromium, checked 17 November 2026). WebKit took the same path nine years later: `git log --diff-filter=A -- Source/WebKit/GPUProcess` in WebKit's repository shows the GPU process added on 4 December 2019 (commit `2d9d0c575ad`), with 2,676 commits touching it by 17 November 2026 (HEAD `f31b755f63d`), 483 of them in the last year.

THE UNSEEN ANGLE

The sandbox switch for an embedded runtime can live outside the app. Microsoft documents that WebView2 reads browser flags from the `WEBVIEW2_ADDITIONAL_BROWSER_ARGUMENTS` environment variable and from registry keys, for local testing and diagnosis, and that `no-sandbox` disables the sandbox for all normally sandboxed process types. Elevated apps ignore those environment flags; ordinary desktop apps do not. A variable left on a branch image after a support session can therefore weaken the teller app, and any other app on that PC that uses the runtime, without a code change or a release. Configuration baselines check for it only if someone adds the rule. Source: Microsoft Learn, "WebView2 browser flags".

This hop in the Frame Ledger

Halbrook's teller desktop on a branch PC, with WebView2 in evergreen mode:

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
1 to 6 Parse to raster: the renderer for the payment screen	Chromium upstream; Microsoft ships it in the runtime	Microsoft, through the evergreen runtime; active after the app restarts	Renderer: Untrusted integrity, job object, win32k lockdown	The browser process, through IPC	**RENT**, **ISOLATE**: keep Site Isolation on
7 Engine compositor: GPU process, command buffer service and `viz`	Chromium upstream; Microsoft ships it	Microsoft, through the runtime; GPU driver by the GPU vendor	GPU process: Low integrity, weaker than the renderer	Shared by all renderers in the process group	**ISOLATE**, **RENT**: never pass `no-sandbox` (Microsoft's list) or Chromium's `disable-gpu-sandbox`
0 Source and host: browser process, teller app and native bridge	Microsoft (runtime) and Halbrook (app)	Microsoft for the runtime; Halbrook for its app	Medium integrity, not sandboxed	The app, and anything the app exposes to script	**OWN**: limit the native bridge to the calls the screen needs
1 to 7 Configuration: browser flags in code, environment variable, registry	Halbrook	Halbrook, through its build and desktop baseline	Applies to all processes in the group	Anyone who can set a variable on the PC	**OWN**: block flags in the build; alert on the variable in the baseline
1 to 7 Process groups: user data folders on VDI	Halbrook	Halbrook	One browser and GPU process per folder	Not applicable	**OWN**: one shared folder per user where the apps allow it

SECURITY LENS

Who can change it? The engine vendor sets the sandbox and process defaults. The embedding app can weaken them with flags or packaging options. On Windows, anyone who can set an environment variable or registry key on the PC can pass WebView2 browser flags to non-elevated apps. The GPU vendor writes the driver the GPU process must trust.

How fast can it be fixed? A renderer or GPU process fix in WebView2 reaches Halbrook as fast as the evergreen update installs and the teller app restarts. A bundled engine, as in Larkfield's app, is fixed as fast as the vendor ships a new build. A GPU driver flaw is fixed at the GPU vendor's pace and then at the fleet's driver-rollout pace.

What does AI change here? The command buffer and IPC interfaces are exactly the narrow, well-specified boundaries that automated and AI-assisted fuzzing explores well. Our view (as of November 2026): expect more reported bugs at the renderer-to-GPU boundary, and treat sandbox configuration drift on embedded runtimes as a monitored control, not a one-time review.

BUSINESS IMPLICATIONS

Cost. Isolation costs memory. Chromium's own numbers are about 10 to 13 percent on desktop with many sites isolated. Halbrook pays it per app process group, per session, on every VDI host. Larkfield pays it on 400,000 customer desktops for its bundled engine. Corvane's Android tablets get no Site Isolation in the system WebView today, which is cheaper and weaker.

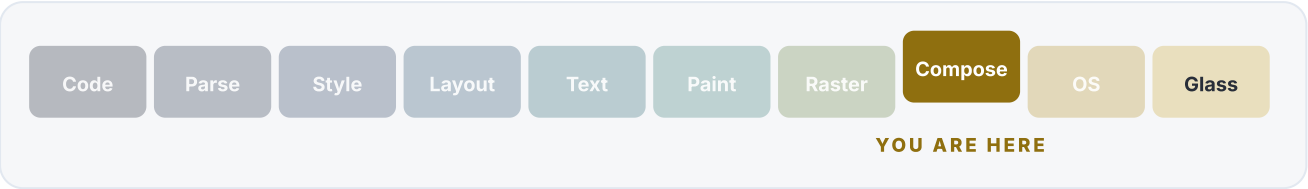
Visibility. Process counts surprise VDI owners. Use the runtime's Browser Task Manager (WebView2's `OpenTaskManagerWindow`) and the `ProcessFailed` event to see which processes exist and when one crashes, and log the browser arguments each app starts with.

Value. The process split is also a reliability feature: a renderer crash no longer has to take down the teller app, if the app handles it. Kept intact, the sandbox turns many renderer bugs into a reload instead of a breach.

Monday rule: search your code, installers and desktop baselines for `no-sandbox`, `disable-gpu-sandbox`, `disable-site-isolation-trials`, `sandbox: false` and `WEBVIEW2_ADDITIONAL_BROWSER_ARGUMENTS`, and make each hit someone's named decision.

CHAPTER EIGHT

Input and Hit-Testing: Who Gets the Click



Every consent screen you ship rests on one quiet assumption: the thing the user touched is the thing your code thinks was touched. That assumption is the output of a chain of hit-tests run by the operating system, the compositor and the engine, and each link can be fooled by a window drawn on top or by input that no human produced. The decision this chapter changes is simple to state: which of your screens must refuse input when they are covered, and which must prove that a person, not a program, pressed the button.

The frame, run backwards

Part I followed a frame from code to glass. Input travels the same road in the other direction. A finger lands on a touch panel. The **digitizer**, the sensor layer under the glass, reports coordinates to a driver. The operating system turns those coordinates into an event and must decide which window owns that point on the screen. That decision is a **hit-test**: a walk through the layers on screen, top first, until one claims the point.

On a desktop the system compositor knows where every window sits, so the OS does the first hit-test. If the window belongs to a web or cross-platform engine, the engine runs a second hit-test inside its own scene: which frame, which element, which handler. Only then does your code run. Each step uses geometry that was computed for drawing. Hit-testing reuses the layout and paint results from Part I and asks them a new question: what is under this point?

That reuse is why the reverse path inherits the forward path's owners. The OS vendor owns the first hit-test. The engine vendor owns the second. You own only the handler at the end, and the rules you set for when it may run.

The message loop and thread affinity

On Windows the reverse path still runs through a design from the 1980s. Microsoft's current documentation describes it plainly: "The system maintains a single system message queue and one thread-specific message queue for each GUI thread." Drivers place mouse and

keyboard input in the system queue. The system then "posts them to the message queue of the thread that created the destination window." The application's **message loop**, a short `GetMessage`, `TranslateMessage`, `DispatchMessage` cycle, pulls each message off and hands it to the window procedure. Modern Windows stacks still sit on this queue: WPF's dispatcher and WinUI 3 desktop windows run a Win32 message loop underneath, and a Chromium-based runtime's browser process receives OS input through its own window's queue before routing it to a renderer.

The consequence is **thread affinity**: a window belongs to the thread that created it, and its input arrives on that thread. Mainstream frameworks impose a similar rule for UI objects. Cocoa and UIKit expect UI work on the main thread, Android expects it on the main (UI) thread, and web engines run your event handlers on the renderer's main thread. Rendering may happen on other threads, as Part I showed with compositor and raster threads, but the handler that decides "approve" usually runs on one busy thread.

For a CTO this matters in two ways. First, a long task on that thread delays every click, so a slow handler feels like a broken button. Second, input routing is a security boundary. Windows Vista added **User Interface Privilege Isolation** (UIPI): a process at a lower integrity level cannot send most window messages to a process at a higher one. Microsoft's archived Vista guidance says such messages are silently dropped. UIPI is why ordinary software cannot drive an elevated administrator window by posting fake clicks to it.

Two hit-tests inside one window

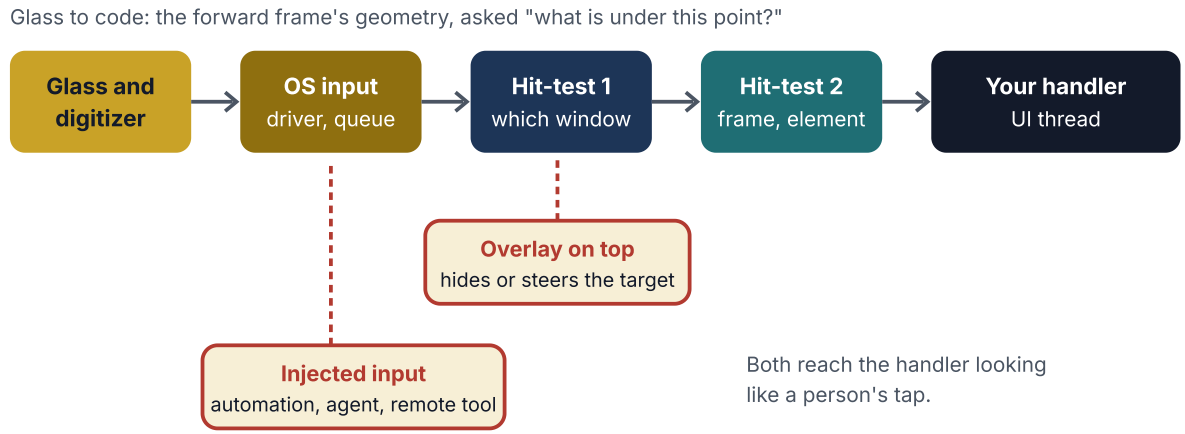
Inside a Chromium-based runtime the browser process receives the OS event first. To decide which frame should get it, including frames from other sites running in other renderer processes, Chromium's `viz` component keeps hit-test region lists submitted with each compositor frame. The source tree under `components/viz/service/hit_test/` holds the aggregator and manager that combine them. When those regions are not enough to decide, Chromium falls back to asking the renderer asynchronously, which is slower.

Then Blink, in the renderer, walks its own layout and paint data to find the target element and dispatch the DOM event. The web adds one marker that native stacks rarely expose: the DOM standard's `isTrusted` flag "indicates whether an event is dispatched by the user agent (as opposed to using `dispatchEvent()`)". A click synthesized by page script carries `isTrusted = false`. A click injected at the OS level, by an automation tool or an agent, enters through the same pipe as a finger and arrives trusted.

Windows does record injection, but only at a level most applications never look. A low-level mouse hook receives an `LLMHF_INJECTED` flag, and `LLMHF_LOWER_IL_INJECTED` when the injecting process ran at a lower integrity level. Your web handler sees neither.

FIGURE 8.1

The reverse path, and where a fake click enters



The OS and the engine each run a hit-test before your code runs. An overlay attacks the geometry; injected input skips the glass entirely.

Clickjacking, tapjacking and overlays

If the hit-test trusts geometry, the attack is to change what the user sees without changing what the hit-test finds. Jeremiah Grossman and Robert Hansen coined the term **clickjacking** in 2008, as the OWASP testing guide records: a hostile page loads a target site in a transparent frame and places its own bait underneath, so the user's click lands on the hidden target. The browser's hit-test is correct. The user's model of the screen is not.

The web's fix was to let a site refuse to be framed. Internet Explorer 8 introduced the `X-Frame-Options` header in January 2009; RFC 7034 described it in October 2013. Content Security Policy's `frame-ancestors` directive is the standard successor. Both protect the page only if the server sends them, which is why the OWASP testing guide still tests for them.

Mobile moved the same trick from frames to windows. An app with permission to draw over other apps can place a window above a banking or settings screen. Yanick Fratantonio and colleagues showed at the IEEE Symposium on Security and Privacy in May 2017, in "Cloak and Dagger", that combining the overlay permission with an accessibility service gave an app full control of the user interface feedback loop, and none of the 20 people in their study noticed the attack.

Android's answers came in layers, each tied to a release. A view can set `filterTouchesWhenObscured`; Android's reference says the framework then discards touches "whenever the view's window is obscured by another visible window at the touched

location". Android 12 made blocking the default for everyone: the system now "prevents apps from consuming touch events where an overlay obscures the app in an unsafe way", with exceptions for accessibility, input method and assistant windows, invisible windows, fully transparent windows, and "sufficiently translucent system alert windows": the system considers "a set of system alert windows" sufficiently translucent, and lets the touch through, "when the combined opacity is less than or equal to the system's maximum obscuring opacity for touches", which is "0.8 by default" in Android 12. The threshold applies to that combined opacity, not to each overlay alone, and windows of type `TYPE_APPLICATION_OVERLAY` are not trusted. That default is a platform behavior of Android 12 and later; it was not backported, so on older devices an app must set `filterTouchesWhenObscured` itself. Android 12 also added the `HIDE_OVERLAY_WINDOWS` permission, which lets an app hide third-party overlays while it is visible. Google's fraud-prevention guide describes it as protection against cloak-and-dagger attacks.

Windows has the strongest version of the idea for one screen only. When User Account Control asks for elevation, the default policy, "Switch to the secure desktop when prompting for elevation", moves the prompt to a separate desktop that ordinary applications cannot draw on or send input to. It is the clearest example of an operating system protecting a consent click by owning the whole frame around it. No equivalent exists for your payment dialog.

Question	Windows	macOS	iOS	Linux	Android
Who routes input to a window	win32k message queues per GUI thread	WindowServer, then `NSView` hit-testing	UIKit `hitTest` in the app's window	The Wayland compositor (X11 lets clients snoop and inject)	InputDispatcher in `system_server`
Can another app cover yours	Yes, topmost windows	Yes, floating windows	Not over another app's screen	Compositor policy decides	With the overlay permission
Defense you can switch on	Secure desktop for UAC only; UIPI between integrity levels	None standard for app windows	System alerts drawn by the OS	None standard	`filterTouchesWhenObscured`, `HIDE_OVERLAY_WINDOWS`; blocking by default since Android 12
Can the app tell injected input	Only through low-level hooks (`LLMHF_INJECTED`)	Not through a standard app API	Not through a standard app API	Not on Wayland clients	Not for accessibility gestures

Read the last row twice. On no platform does a typical application handler receive a reliable "a human did this" signal. That is the gap the next two chapters widen.

HOW IT GOT THIS WAY

Win32 kept a message-queue model in which input is delivered to the thread that owns the window, and Windows Vista added privilege rules on top: UIPI and the secure desktop for UAC. The web began with no framing rules at all; clickjacking forced `X-Frame-Options` in 2009 and `frame-ancestors` later. Android allowed overlays for years because legitimate apps and accessibility tools use them, then tightened the rules release by release, reaching default blocking of untrusted touches in Android 12 (2021). Each fix protected a named attack, not the general question of whether the user saw what they clicked.

What hasn't changed: hit-testing still walks the same geometry that was painted, and Microsoft's current message-queue page (updated July 2025) still describes the per-thread queue and the `GetMessage` loop that Win32 programs have used since the 1990s.

The click an agent makes

Automation is old: test tools, remote support and robotic process automation have injected input for decades. What changed is volume and judgment. A software agent that acts for the user now reaches your handler in one of two ways, and neither looks different from a person at the handler. **Injected input**, such as plain `SendInput`, Android's `AccessibilityService.dispatchGesture` and the Chrome DevTools Protocol's input commands, travels the same path as a person's pointer or touch and arrives as ordinary events. **Semantic actions**, such as UI Automation's Invoke pattern, skip the pointer path entirely: the client asks the control's provider to "activate a control and initiate its single, unambiguous action", in Microsoft's words, and the control runs the action a click would have run. Neither carries proof that a person meant it.

This turns three habits into risks. Logging a click as "user approved" claims more than the log can prove. Relying on a dialog as consent assumes a person read it. And testing a screen only with human testers misses how an automated reader will see a covered or half-drawn button. Chapter Ten covers how agents read the screen; this chapter's point is narrower: the input pipe has no field that says who pressed.

THE UNSEEN ANGLE

An agent's injected click arrives through the same pipe as a user's, and an agent's semantic action through UI Automation skips the pipe and calls the control directly. The one platform marker for injected input is usually thrown away, and a semantic action never produces pointer input to mark. Windows sets `LLMHF_INJECTED` on injected mouse events, but only a low-level hook sees it; the web's `isTrusted` flag catches script-made clicks but not OS-injected ones. So on most desktops nothing in the frame or the log says which one approved the payment. Where a screen must show that the enrolled person was present, ask the platform for it: an OS-level user-verification prompt such as Windows Hello, or a passkey with WebAuthn user verification (a biometric or PIN on the authenticator). Even that proves presence, not intent. Sources: Microsoft Learn, MSLLHOOKSTRUCT and IInvokeProvider::Invoke; WHATWG DOM Standard, `isTrusted`; W3C Web Authentication Level 3, "User Verification".

> *The hit-test is right about geometry. The question is whether the user and the hit-test were looking at the same screen.*

This hop in the Frame Ledger

Corvane Hotels leads this chapter. Its check-in kiosks run a Windows IoT shell with Microsoft's Edge WebView2 runtime, which is Chromium-based; guests tap "Confirm stay and charge card". Its front-desk Android tablets approve room moves and refunds.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
Input side row: touch digitizer and Windows pointer stack on the kiosk	Panel vendor firmware; Microsoft for the OS stack; Corvane pays through hardware and licences	Windows Update monthly; panel firmware rarely, by field visit	Kernel driver, then win32k in the kiosk session	Any process in the session can inject at its integrity level	**RENT**, **ISOLATE**: single-app kiosk session, no remote-support tool resident
Input side row, hit-test at 7 Engine compositor: Chromium `viz` and Blink event targeting	Upstream Chromium; Microsoft ships it in WebView2	Runtime channel; evergreen or pinned per Chapter Eleven	Browser, GPU and renderer processes	DevTools protocol if a debug port is open	**RENT**, **PIN**: close remote debugging in production images
Input side row, handler: "Confirm stay and charge card"	Corvane's kiosk team	Corvane's own release train	Renderer, sandboxed	Corvane telemetry	**OWN**: accept only `isTrusted` events, which rejects script-made clicks but not OS-injected input; require a second, separated confirm for the charge
Input side row: front-desk tablet overlay defenses	Google and the tablet OEM; Corvane's app team sets the flags	Monthly bulletins, then OEM delay	InputDispatcher in `system_server`	Accessibility services can still dispatch gestures	**OWN**, **ISOLATE**: set the flags (`HIDE_OVERLAY_WINDOWS` , `filterTouchesWhenObscured`); MDM blocks overlay grants

SECURITY LENS

Who can change it? The OS vendor decides which windows may cover yours and which input may be injected. The engine vendor decides how frames and elements are hit-tested. You decide only whether your handler refuses obscured or synthetic input, and the operator of the device decides which overlay and accessibility apps are installed.

How fast can it be fixed? Overlay rules change with OS releases: Android's default blocking needed Android 12, so tablets that cannot upgrade never get it. Your own flags ship with your next app release. Header-based web defenses ship with a server change in hours.

What does AI change here? Agents make injected input normal rather than rare. A defense that treats all injected input as hostile now breaks legitimate assistive and agent use, and one that accepts it all lets a hijacked agent approve anything. Our view (as of November 2026): expect to need a policy per screen, not per device.

BUSINESS IMPLICATIONS

Cost. On Corvane's 3,000 kiosks, a mis-routed tap is not a bug ticket but a guest at a dark desk and a field visit. Halbrook Bank's teller desktops pay differently: a blocked UI thread on the approval screen makes tellers double-click, and duplicate submissions cost reconciliation time.

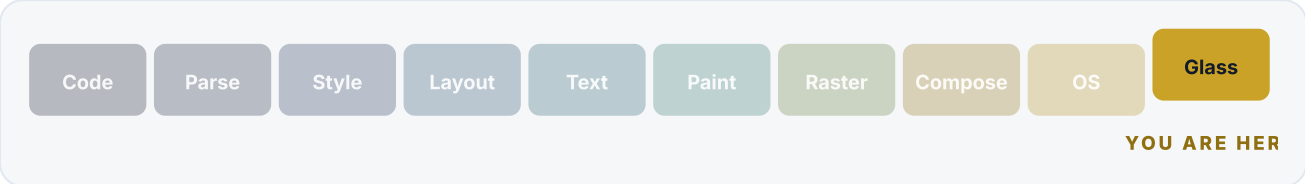
Visibility. Synthetic and human clicks look the same in application logs. Larkfield Software cannot tell its customers which share of actions in its app came from automation unless it records the signals the platform offers, where they exist, and says plainly where they do not.

Value. Input latency is felt before any frame is judged, and a consent click you can defend is worth more than one you can only log. Screens that refuse covered input and separate the decisive tap from the rest earn trust in audits and in disputes.

Monday rule: list the five screens where a click moves money or grants consent, and for each write down whether it refuses obscured input and whether it can tell injected input. Fix the first "no".

CHAPTER NINE

Capture, Remote Display and the Trusted Pixel



Somewhere in your company a dispute is settled with a screenshot: "this is what the customer saw when they approved." That screenshot was produced by an operating system API you did not choose, from a copy of the screen that may differ from what the panel showed. The decision this chapter changes is which screens must resist capture, against which threat, and what you keep as evidence of what was displayed, because a picture of the screen is not that evidence.

Every copy of the frame goes through the OS

Chapter Five ended with the system compositor building one desktop image. Everything that copies the screen copies from that point or near it, and each operating system offers its own official doors.

Windows has two. The **Desktop Duplication API** arrived with Windows 8, when, in Microsoft's words, Windows 8 "disables standard Windows 2000 Display Driver Model (XDDM) mirror drivers and offers the desktop duplication API instead." It hands a whole monitor's image to the caller as a GPU surface, with lists of dirty and moved rectangles. **Windows.Graphics.Capture** followed in Windows 10 version 1803. Its picker path is consent-based: the user picks a window or display in a system picker. It is not the only path. Since Windows 10 version 1903, the Win32 interop interface `IGraphicsCaptureItemInterop` lets a desktop app create a capture item directly from a window handle (`CreateForWindow`) or a monitor handle (`CreateForMonitor`), with no picker. On both paths, by default, "a yellow notification border is drawn by the system around the actively captured item." The border can be turned off: from Windows build 20348 (Windows Server 2022 and Windows 11), `GraphicsCaptureSession.IsBorderRequired` can be set to false, but only after the user accepts a separate borderless-capture prompt, the app declares the `graphicsCaptureWithoutBorder` capability, and no other app capturing the same item requires the border.

macOS uses **ScreenCaptureKit**, introduced in macOS 12.3 and presented at WWDC22, gated by the user's Screen Recording permission. **iOS** gives apps no general way to read other apps' screens; system screen recording and ReplayKit broadcasts are user-started. **Linux** splits by display server. Under X11 any client can read the whole screen. Under Wayland, apps ask the `xdg-desktop-portal` ScreenCast interface, which "typically presents a dialog to the user" and returns **PipeWire** streams. **Android** uses `MediaProjection`, added in API level 21 (Android 5.0), with a consent dialog; Android 14 QPR2 (March 2024) let users share a single app window instead of the whole screen. Google's MediaProjection guide credits the feature to Android 14 (API level 34); its Android 14 feature page and the March 2024 announcement place it in QPR2.

The pattern is the same everywhere: the OS owns the composed frame, so the OS decides who may copy it. Your app is a guest in that frame.

What the copy leaves out, and what it adds

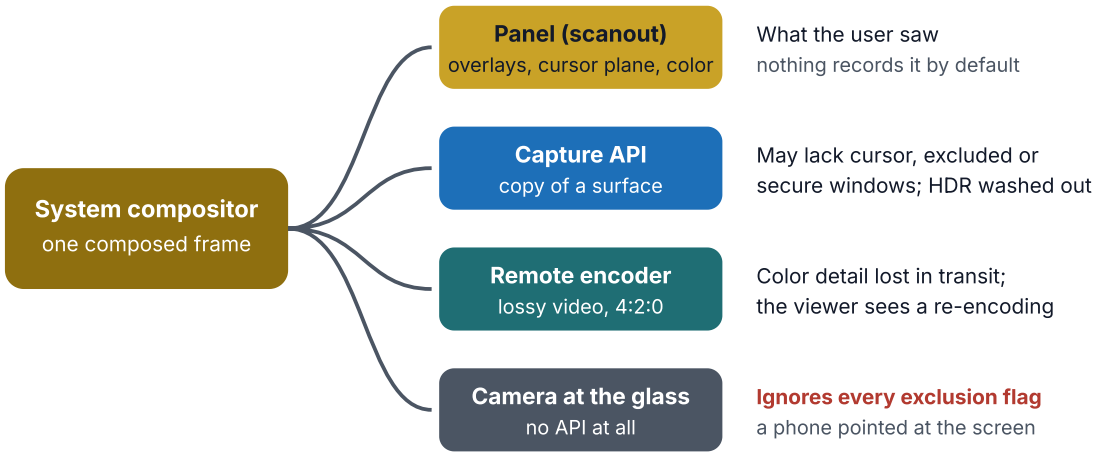
A screenshot is usually a copy of a composed surface, not a photograph of the glass. The copy can differ from what the panel showed for reasons that are documented, not exotic.

- **The cursor.** Microsoft's Desktop Duplication guide says the mouse pointer is either drawn into the image or overlaid separately by the graphics adapter, in which case "your client app must draw the mouse pointer shape onto the desktop image to accurately represent what the current user will see." A capture tool that skips this step records a screen with no pointer.
- **Excluded and secure windows.** A window marked for exclusion is missing from the capture; on Android a `FLAG_SECURE` window appears blank. The panel still showed it.
- **Color and HDR.** Microsoft warns that capturing HDR content into the default 8-bit format causes "pixel overclipping (i.e. the captured content looks washed out)". Color management after composition means the panel's light and the file's numbers are not the same thing.
- **Time.** Duplication delivers frames when they change and may coalesce updates. A capture shows one frame; the user saw a sequence, including a toast or a dialog that came and went between captures.
- **Remote encoding.** A remote session's viewer sees a re-encoded image. Azure Virtual Desktop's documentation says the default chroma value is 4:2:0, "a good balance between image quality and network bandwidth"; fine colored text loses detail before it reaches the remote screen.

None of this makes capture useless. It makes it a record of what one API returned at one moment, which is weaker than "what the user saw".

FIGURE 9.1

One composed frame, four different records



Four records start from one composed frame: the panel, a capture API copy, a remote session's encoded video and a camera aimed at the glass. Each ends with different pixels. Only the panel's version is what the person saw, and the camera records the panel, not the frame.

Exclusion flags and their threat models

Each platform offers an app a way to keep a window out of copies. They are not the same control, and the honest way to compare them is by the threat each one stops.

Windows `SetWindowDisplayAffinity`. Since Windows 7 an app could set `WDA_MONITOR`, which shows the window "only on a monitor. Everywhere else, the window appears with no content." Windows 10 version 2004 (2020) added `WDA_EXCLUDEFROMCAPTURE`: "Everywhere else, the window does not appear at all." Microsoft is explicit about the limit: "It is important to note that unlike a security feature or an implementation of Digital Rights Management (DRM), there is no guarantee that using SetWindowDisplayAffinity and GetWindowDisplayAffinity, and other necessary functions such as DwmIsCompositionEnabled, will strictly protect windowed content, for example where someone takes a photograph of the screen." So it is a control with a limited threat model, not a security boundary. It keeps the window out of the public capture APIs used by ordinary recording, meeting and screenshot tools. It does not stop a camera, a remote session that sends the screen by other means, or code running with enough privilege to read memory or the driver.

Android `FLAG_SECURE`. The flag, present since API level 1, tells the system to treat the window "as secure, preventing it from appearing in screenshots or from being viewed on non-secure displays." Google's fraud-prevention guide adds that a screenshot of such a

window "is blank" and that the flag helps against background screenshots and abused screen-sharing apps. This is an enforced control: the system, not the app, blanks the copy. Its threat model still stops at the device boundary. It does not stop a camera, it does not stop an accessibility service from reading the view tree (Chapter Ten), and it cannot help on a compromised device. Google's own guide notes it "isn't reliable in preventing overlay attacks."

macOS `NSWindow.sharingType`. Setting it to none asks the window server to keep the window out of shared content. The evidence here is thin and should be read as such: developer reports from 2025 say windows set this way still appeared in ScreenCaptureKit captures on macOS 15.4 and later, and the one Apple reply on the Apple Developer Forums, from a developer technical support engineer, said: "At this time there are no public APIs for preventing screen capture." Treat it as a request, not a control.

iOS and Linux. iOS lets an app detect capture (`UIScreen.isCaptured` since iOS 11) rather than exclude itself. A common workaround places sensitive views inside the internal layer of a text field with `isSecureTextEntry` set, which the system hides from captures; it relies on undocumented view structure that has changed between iOS versions, so treat it as a workaround, not a control; Android 14 and 15 added similar detection callbacks for screenshots and screen recording. Linux has no standard exclusion: under Wayland the user's portal choice is the control, under X11 there is none.

Question	Windows	macOS	iOS	Linux	Android
Official capture door	Desktop Duplication (Windows 8); Windows.Graphics.Capture (1803)	ScreenCaptureKit (12.3)	User-started recording, ReplayKit	Wayland: ScreenCast portal to PipeWire; X11: any client	`MediaProjection` (API 21)
User consent shown	Graphics.Capture: picker, or no picker through Win32 interop (1903); yellow border by default on both. None for Duplication	Screen Recording permission	System indicator	Portal dialog	Consent dialog per session
App-side exclusion	`WDA_EXCLUDEFROMCAPTURE` (Windows 10 version 2004, 2020)	`sharingType` none, not honored by all capture paths	Detection only	None standard	`FLAG_SECURE`
Vendor's own wording	"no guarantee"	"no public APIs for preventing screen capture"	Not applicable	Not applicable	Enforced against screenshots and non-secure displays

The table changes one decision: a regulated screen that must run on several platforms cannot promise the same capture protection on all of them, so the evidence of what was approved must not depend on it.

The trusted pixel: what can be proved

If capture cannot prove what the user saw, can anything? On supported Android devices, yes, within a narrow scope. **Android Protected Confirmation**, available from Android 9 (API level 28), shows a short prompt through a system-controlled, non-customizable full-screen dialog. Android's reference says a positive response "indicates with high confidence that the user has seen the given text, even if the Android framework (including the kernel) was compromised", and that this "requires dedicated hardware-support and may not always be available." The app signs the confirmed message with a Keystore key created with `setUserConfirmationRequired`, and the bank's server checks the signature, the key's attestation chain and a fresh nonce.

Its limits matter as much as its strength. Only the prompt text is confirmed: Android warns that "the `promptText` is the only part of the message that the user actually confirmed". It is not a confidential channel. It proves the text was displayed and a confirm action was taken, not that a person understood it. Prompts may be unavailable while accessibility services run, and `ConfirmationPrompt.isSupported` must be checked on each device; where it returns false, server-side dynamic linking (below) is the control that remains. Android's developer pages still document the API, with no deprecation notice (checked 18 November 2026). It is one strong, scoped claim, not display attestation in general.

Regulation asks for something weaker and different. The EU's PSD2 strong customer authentication rules, Delegated Regulation (EU) 2018/389, Article 5, require **dynamic linking**: "the payer is made aware of the amount of the payment transaction and of the payee", the authentication code is "specific to the amount of the payment transaction and the payee", and "any change to the amount or the payee results in the invalidation of the authentication code generated." Article 5(2) asks payment providers to protect the confidentiality, authenticity and integrity of the amount and payee and of the information displayed to the payer through all phases of authentication. That is a requirement to show the right facts and bind the code to them. It is not a requirement, or a mechanism, to attest which pixels reached the glass.

Remote display, briefly

Virtual desktops and remote sessions add another copy. The session host renders the frame, a remote display protocol such as RDP or Citrix HDX sends drawing commands, cached images or a video-encoded stream, and the client decodes and composes it again. For this

chapter the point is evidence: the advisor's screen at home is a re-encoding of the frame, subject to the codec's color and timing, and capture on either end records a different copy. The cost side, encoders, bandwidth and GPU sharing, belongs to Chapter Fifteen.

HOW IT GOT THIS WAY

Composited desktops made capture cheap: once Windows Vista and Mac OS X kept every window in an off-screen buffer, copying the screen became reading a surface. Windows 7 added display affinity for content protection; Windows 8 replaced mirror drivers with Desktop Duplication in 2012; Windows 10 version 1803 added Graphics.Capture with a consent picker in 2018, and version 1903 added a Win32 path that needs no picker. Android shipped `FLAG_SECURE` from its first API level and `MediaProjection` in Android 5.0 (2014). Wayland's portals gave Linux a consent model that X11 never had. Book 1 Chapter Thirteen covered how an OS feature that records the screen turns capture into a data store; that history is not repeated here. The decision today is Halbrook's: which screens forbid capture, and what it keeps instead.

What hasn't changed: the OS, not the app, controls the final pixel and every official copy of it. Microsoft's display-affinity page still says the feature "works only when the Desktop Window Manager (DWM) is composing the desktop", which since Windows 8 is all the time.

THE UNSEEN ANGLE

The app does not own the final pixel, and the strongest "proof of display" on the market proves less than people assume. Android Protected Confirmation binds a signature to the prompt text only, and Android tells relying parties never to assume that the extra data they sent "corresponds to the `promptText`". A team that puts the amount in the extra data and a generic "Confirm payment?" in the prompt has a signature over the wrong thing. Source: Android Developers, Android Protected Confirmation guide and `ConfirmationPrompt` reference.

> *A screenshot is a copy of a surface. The dispute is about the glass.*

This hop in the Frame Ledger

Halbrook Bank leads this chapter. Its branch payment-approval screen runs in a Windows desktop app with Microsoft's Edge WebView2 runtime; its mobile app approves payments on iOS and Android; advisors work on VDI.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
Capture side row: Desktop Duplication and Graphics.Capture on branch PCs	Microsoft; Halbrook through Windows licensing	Windows Update, monthly	`dwm.exe` and the calling process	Any recorder, meeting or support tool in the session	**RENT**, **ISOLATE**: application control allows only approved capture tools on teller PCs
Capture side row, exclusion: `WDA_EXCLUDEFROMCAPTURE` on the approval window	Halbrook sets it; Microsoft implements it	Halbrook's app release	Kernel-mode window state	Cameras, remote sessions, privileged code	**RENT**: document as a control with a limited threat model: it blocks the public capture APIs, not cameras, remote sessions or privileged code
Capture side row, remote display: advisor VDI session	VDI vendor and Microsoft; Halbrook pays per seat	Vendor release cycle	Host encoder, client decoder	Remote client, both endpoints' capture tools	**RENT**, **ISOLATE**: exclude payment approval from VDI or require mobile confirmation
10 Glass and readers: mobile approval with `FLAG_SECURE` plus Protected Confirmation where `isSupported`	Google and device makers; Halbrook writes the prompt	Monthly bulletins plus OEM delay; prompt text with app release	System dialog backed by secure hardware	The user; accessibility services may block the prompt	**OWN**: the prompt text, with amount and payee in `promptText`, verified on the server
10 Glass and readers: evidence of what was shown	Halbrook	Halbrook	Server side	Auditors, dispute teams	**OWN**: store the rendered amount and payee strings, the authentication code binding and any signed confirmation; screenshots only as supporting material

SECURITY LENS

Who can change it? The OS vendor decides who may capture and what an exclusion flag means, and can change it in a release, as the macOS 15 developer reports suggest. You decide which flags to set and what you keep as evidence. The device owner decides which recording tools are installed.

How fast can it be fixed? A flag ships with your next release. A change in how the OS honors it arrives on the OS vendor's schedule and may never reach older devices. Hardware-backed confirmation depends on the device fleet you already have.

What does AI change here? Screen-reading agents are capture clients with judgment. They use the same official doors, and a window excluded from capture may still be readable through the accessibility tree. Exclusion flags were designed for recorders, not for software that reads and acts.

BUSINESS IMPLICATIONS

Cost. Halbrook's real cost is in disputes: a case settled on a screenshot that lacked the cursor or the dialog that mattered is a case that can be reopened. Corvane pays in a different way: kiosk screens that show card details need exclusion on Windows IoT and a policy for the support tools its field team uses.

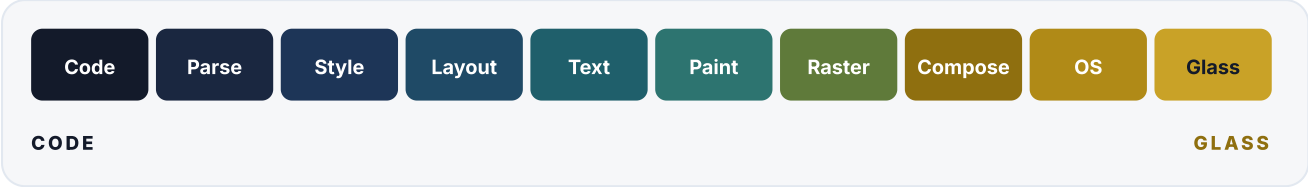
Visibility. A screenshot is a copy, not a record. Larkfield Software, whose customers record its app in meetings, should state in its documentation which windows are excluded from capture and on which platforms the flag is honored, so customers do not assume protection they do not have.

Value. Consent and payment flows depend on a display that nobody attests in general. A flow that binds the amount and payee to the authentication code, and uses hardware confirmation where available, is a product feature that regulators and customers can check.

Monday rule: for the payment-approval screen, write down what evidence you would produce in a dispute. If the answer is a screenshot, add the server-side record of the exact strings displayed and the code that was bound to them.

CHAPTER TEN

When the User Is an Agent



Your screens now have a second audience: software that reads them and acts for a person. It does not see the glass. It reads a captured image, the page's DOM or the operating system's accessibility tree, and those three can disagree, by accident or because someone arranged it. The decision this chapter changes is what your critical screens expose to an agent, which representation an agent may trust, and whether you test that all three say the same thing about the amount, the payee and the consent text.

Three screens behind one screen

A person sees light. Software that wants to "see" the same screen has three representations to choose from, each produced at a different hop of the frame.

Pixels. The agent takes a capture through the official doors of Chapter Nine and reads it with a vision model or **optical character recognition** (OCR), software that turns an image of text into text. Pixels are the closest to what the user saw, but they carry no structure and no origin: a button drawn by your app and a button drawn by an overlay look the same.

The DOM. For web and embedded-engine apps, an agent with access to the engine (through an automation protocol or a browser extension) can read the **Document Object Model**, the tree of elements the page built in Chapter Two. The DOM holds text and attributes, including text that is never drawn.

The accessibility tree. The engine or UI framework publishes a simplified tree of roles, names, states and actions to the operating system, for screen readers and other assistive technology. The W3C's Core Accessibility API Mappings specification, in its version 1.2 draft (Candidate Recommendation Draft, 5 November 2026), describes how browsers map web content to five platform APIs: MSAA with IAccessible2, UI Automation, ATK/AT-SPI, the macOS accessibility protocol and the Android accessibility API. Earlier versions covered only the first four. Any process the OS allows to be an accessibility client can read the tree and invoke its actions.

The accessibility APIs, and why agents use them

These APIs were built for assistive technology, and they are old. **Microsoft Active Accessibility** (MSAA) shipped in 1997 as an add-on for Windows 95. **UI Automation** followed with Windows Vista and .NET Framework 3.0 in 2006; Microsoft's current page says it "enables assistive technology products, such as screen readers, to provide information about the UI to end users and to manipulate the UI by means other than standard input," and adds that it "also allows automated test scripts to interact with the UI." Apple exposes `NSAccessibility` on macOS and `UIAccessibility` on iOS. Linux desktops use **AT-SPI**, which moved from CORBA to D-Bus during the GNOME 3 transition. Android's `AccessibilityService` dates from API level 4 (Android 1.6); the node tree it reads came later, with `AccessibilityNodeInfo` in API level 14 (Android 4.0) and `AccessibilityWindowInfo` in API level 21 (Android 5.0). Today a service can retrieve window content "represented as a tree of `AccessibilityWindowInfo` and `AccessibilityNodeInfo` objects" and can dispatch gestures to the touch screen.

For an agent, the tree is attractive because it is structured, cheap and already tells it what each control does. When an agent acts through it, its clicks reach your handler like a person's: Chapter Eight showed that the input pipe has no field that says who pressed. For a CTO, that means the accessibility tree has become a de facto agent interface. The labels, roles and hidden nodes you ship are now read by software that makes decisions, not only by people using screen readers.

Computer-use agents made this mainstream. Anthropic released computer use in public beta on 22 October 2024; OpenAI introduced Operator on 23 January 2025 and folded it into ChatGPT agent in July 2025. Both read screenshots and act with mouse and keyboard. Browser automation protocols, such as the Chrome DevTools Protocol, can also hand an agent the DOM.

What an agent reads	Windows	macOS	iOS	Linux	Android
Accessibility tree	UI Automation (MSAA bridged)	`NSAccessibility` (AX API)	`UIAccessibility`, in-app only	AT-SPI over D-Bus	`AccessibilityService`
Pixels	Windows.Graphics.Capture, Desktop Duplication	ScreenCaptureKit	User-started recording only	PipeWire portal; X11 any client	`MediaProjection`
Acting	UIA patterns (semantic), `SendInput` (injected)	AX actions, synthetic events	Not across apps	AT-SPI actions, compositor-specific input	Accessibility actions (semantic), `dispatchGesture` (injected)
Does the capture flag hide the tree	No: display affinity governs display and capture, not UIA	No: `sharingType` governs sharing	Not applicable	No standard flag	No: `FLAG_SECURE` blanks screenshots; Android 14 added `accessibilityDataSensitive` for the tree

The last row is the decision. Locking pixels is not locking the screen.

How the three come to disagree

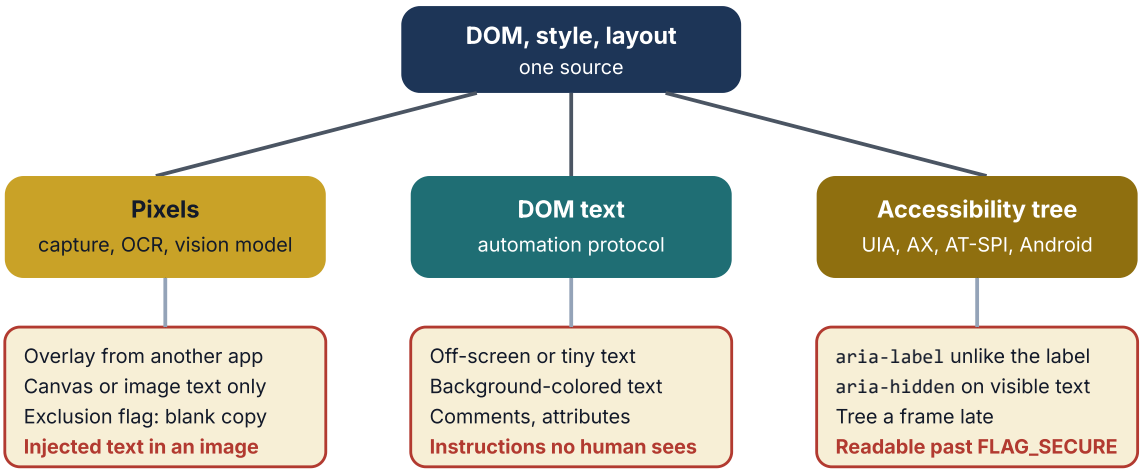
All three representations come from the same source: the DOM, its computed style and its layout. Nothing forces them to agree. The web platform has features whose whole purpose is to make them differ, for good reasons:

- **Hidden from people, visible to software.** Text clipped off-screen for screen-reader users, or styled too small or in the background color, is absent from the pixels but present in the DOM and the tree.
- **Visible to people, hidden from software.** WAI-ARIA's `aria-hidden="true"` removes content from the accessibility tree while it stays on screen. Text drawn into a `canvas` or an image exists only in the pixels unless the author adds a description.
- **Different words.** An `aria-label` replaces the accessible name. WCAG 2.1 added success criterion 2.5.3, "Label in Name" (Level A), because names and visible labels drift apart: when they do not match, "speech-input users who attempt to use the visible text label as a means of navigation or selection" fail.
- **Another window on top.** An overlay from another app (Chapter Eight) changes the pixels but not your app's DOM or tree. An agent reading your tree "sees" a screen the user cannot.
- **Time.** The tree updates on its own schedule. An agent can read a node a frame before or after the paint the user saw.

Each mechanism is legitimate. Together they mean an attacker who controls any content on the page, a product review, an email body, a document preview, a support chat, can show one thing to the person and say another to the agent.

FIGURE 10.1

One layout, three screens, and where they split



The three representations share a source but are produced separately. Every box in red marks a documented way for them to differ, which an attacker can use on purpose.

Prompt injection through the screen

Prompt injection is text that a language model reads as an instruction although it came from content, not from the user. Greshake and colleagues described the indirect form at the ACM AISec workshop in 2023 ("Not what you've signed up for"): an attacker plants instructions in data an LLM-integrated application will later retrieve, and the application acts on them. A screen is that kind of data. Anything an agent reads, pixels, DOM or tree, can carry instructions.

Vendors say so directly. Anthropic's computer-use documentation warns that "instructions on webpages or contained in images might override your instructions or cause Claude to make mistakes," advises developers to "isolate Claude from sensitive data and actions," and describes classifiers that "automatically scan what the tool returns, such as screenshots, to flag potential prompt injections." The defense exists because the channel exists.

The three-screens problem makes injection easier to hide. Text the person cannot see (background-colored, off-screen, in an attribute) is still in the DOM and often in the tree. Text the tree hides can still be in the pixels. A reviewer who checks the screen by eye checks one representation of three.

Capture flags do not protect the tree

Chapter Nine's exclusion flags work on capture. They say nothing about accessibility. On Android, `FLAG_SECURE` blanks screenshots; an accessibility service can still read the window's node tree, and its own screenshot call returns `ERROR_TAKE_SCREENSHOT_SECURE_WINDOW` while the text remains readable as nodes. This is exactly the gap "Cloak and Dagger" exploited in 2017 by pairing overlays with an accessibility service. Android 14 (API level 34) added `setAccessibilityDataSensitive`, which limits a view to accessibility services that declare themselves accessibility tools. It narrows the gap on new devices; it does not close it on older ones.

On Windows, display affinity governs where a window's content can be displayed and captured. Microsoft's documentation for it does not mention UI Automation, and UI Automation is a separate API that reads the provider tree, not the pixels. A teller window excluded from capture can still publish the amount and payee to any UIA client in the session. The same reasoning applies to `sharingType` on macOS. Treat the tree as its own surface, with its own owner and its own policy.

One tempting fix is wrong. Turning the accessibility tree off, or hiding critical screens from it, also locks out the people the tree was built for. Screen-reader users would lose the payment screen, and that breaks accessibility law: the Americans with Disabilities Act in the United States, the EN 301 549 standard used in European public procurement, and the European Accessibility Act, which has applied to many banking and e-commerce services since 28 June 2025. The path is semantic parity: the tree, the DOM and the pixels say the same thing about the facts that matter, and the confirmation that moves money needs a step an agent cannot complete alone.

HOW IT GOT THIS WAY

Software read screens long before language models. Screen-scraping and robotic process automation drove legacy applications by pixels and window handles; test tools used MSAA (1997) and then UI Automation (2006) because those APIs were the only structured view of another program's UI. Apple, GNOME and Android built their equivalents for the same reason: assistive technology needs to read and operate every app. WCAG is referenced in accessibility procurement rules, such as the EU's EN 301 549, so apps sold to large buyers tend to publish a rich tree. Computer-use agents in 2024 and 2025 inherited all of it. The decision today is whether an agent may act on a screen at all, and through which of the three representations.

What hasn't changed: the accessibility tree is derived from the same layout as the pixels, yet nothing in the platform forces them to agree; Microsoft's archive notes that UI Automation in Windows 7 still kept a bridge to Active Accessibility clients.

THE UNSEEN ANGLE

Agents create a second screen, and most teams test only the first. Pixels, DOM and accessibility tree are three screens; a trustworthy UI makes them agree on the critical facts (amount, payee, consent text) and tests that they do, the same way WCAG 2.5.3 already asks that the accessible name contain the visible label. That check is cheap to automate, yet a team rarely finds it in a standard accessibility audit, which tests labels rather than the business facts that matter. Sources: W3C, Understanding SC 2.5.3 Label in Name; W3C Core-AAM 1.2.

> *Locking the pixels is not locking the screen. The tree is a screen too.*

This hop in the Frame Ledger

Larkfield Software leads this chapter. Its customers increasingly run agents that operate Larkfield's desktop app on their behalf, on 400,000 desktops Larkfield does not control. Its "Share report externally" dialog is the critical screen.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
Capture side row: OS capture used by agents	OS vendors; the agent vendor; the customer pays	OS updates; agent vendor's releases	System compositor and the agent process	Any approved capture client	**RENT**: set capture exclusion on the token and secret views, documented as a control with a limited threat model (public capture APIs only)
1 Parse, the DOM: bundled engine's automation and debug ports	Larkfield, as the engine bundler	Larkfield's own release; customers update on their schedule	Renderer and browser processes	Anyone who can reach an open debug port	**OWN**, **PIN**: no remote debugging in production builds; sign and pin the engine
10 Glass and readers, the accessibility tree: Chromium accessibility mapped to UIA, AX and AT-SPI	Chromium upstream for the mapping; Larkfield for roles and names	Mapping with each engine update; semantics with Larkfield's release	Browser process publishes to the OS API	Screen readers, test tools, agents	**OWN**: the semantics; names contain visible labels; amount, recipient and scope appear identically in pixels, DOM and tree, tested in CI
Input side row, agent action: UIA patterns, synthetic input	OS vendors and agent vendors	Outside Larkfield's control	Agent process at user integrity	The agent and its logs	**ISOLATE**: "Share externally" requires user verification by the enrolled person, not the agent: a Windows Hello prompt, or a passkey with WebAuthn user verification (biometric or PIN on the authenticator). It proves the enrolled user was present, not what they intended

SECURITY LENS

Who can change it? You write the DOM and the accessible names. The engine vendor writes the mapping to each OS API. The OS vendor decides which processes may read the tree and act through it, and the device owner decides which agents and accessibility services are installed.

How fast can it be fixed? A mismatch between label and name, or a hidden instruction channel in your own markup, is fixed in your next release. Content you display from others (messages, documents, reviews) cannot be fixed at the source; it must be treated as untrusted on every release. OS-level limits, such as Android 14's data-sensitive flag, reach only devices that update.

What does AI change here? Everything in this chapter. The accessibility tree, built for people who need it, is now also the cheapest way for software to act with a user's permissions, and the screen has become an input channel for instructions. Our view (as of November 2026): plan for agents as a new class of user with its own threat model.

BUSINESS IMPLICATIONS

Cost. For Larkfield, every redesign can break customers' agent workflows the way an API change breaks integrations, so UI semantics now need versioning and release notes. Halbrook Bank faces the opposite cost: blocking agents on regulated desktops means policing accessibility clients without breaking the screen readers its staff rely on.

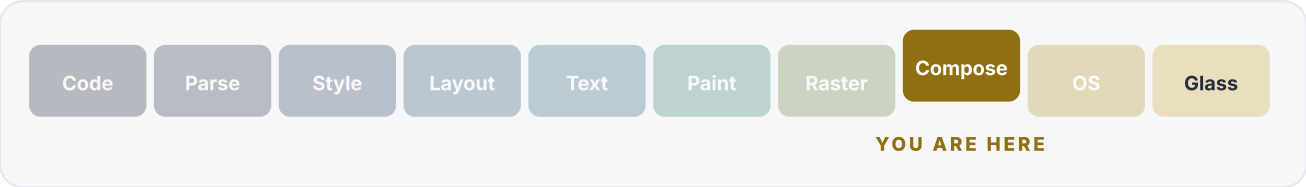
Visibility. Agent actions look human in logs (Chapter Eight), and agent reads leave no trace at all. Corvane Hotels cannot tell whether a front-desk tablet's refund screen was read by staff or by an installed accessibility app unless it inventories accessibility services through device management.

Value. Agent-readable screens become a product feature. An app whose critical facts agree across pixels, DOM and tree is easier to automate safely, easier to make accessible, and easier to defend in an audit.

Monday rule: pick one critical screen and write a test that reads the amount (or recipient, or consent text) three ways, from a capture with OCR, from the DOM and from the accessibility tree, and fails if they differ.

CHAPTER ELEVEN

The Browser Inside the App: Eight Runtimes on One Ledger



When your team picks an embedded web runtime, it is not picking a feature list. It is signing up to an update clock: the schedule on which someone else fixes the next zero-day in the engine that draws your screen, and the steps it takes before that fix is running on your devices. Eight runtimes cover the common enterprise cases (WebView2, Electron, CEF, Qt WebEngine, WKWebView, WebKitGTK, Android System WebView and Tauri), and they fall into four decisions: an evergreen engine shared by every app, an engine bundled inside your own app, an engine tied to the operating system, or no web engine at all. This chapter puts them on one Frame Ledger so the choice can be made on who patches, how fast, and at what cost per seat.

Four decisions, not eight products

An **embedded web runtime** is a browser engine without the browser around it: no address bar, no tabs, just the parse, style, layout, paint and compositing machinery from Part I, exposed to your app as a control. The products have different names, but the decision underneath is about who owns the engine's clock.

- **Evergreen and shared.** One engine is installed per machine, updated by its vendor, and used by every app that asks for it. Microsoft's Edge WebView2 runtime in its evergreen mode is the main example.
- **Bundled.** Your installer carries its own copy of the engine, and you decide when it moves. Electron, the Chromium Embedded Framework (CEF), Qt WebEngine and WebView2's fixed-version mode all work this way.
- **OS-tied.** The engine is part of the operating system and moves when the OS or its update channel moves: Apple's WKWebView, WebKitGTK as packaged by a Linux distribution, and Android System WebView. Tauri, through its `wry` library, deliberately rents these engines instead of shipping one.

- **Native.** No web engine draws the screen. That path is the subject of Chapter Twelve.

Each decision moves the same three costs to a different owner: the patch work, the memory on every seat, and the compatibility testing. None of them removes those costs.

Evergreen and shared: WebView2

Start with the point fleet reports most often miss: an installed fix is not a running fix. An evergreen update installs on the vendor's clock, but it takes effect only on the app's clock, when the app releases the old engine or restarts. Everything below follows from that.

WebView2 runs on the Edge browser's process model with "modified Microsoft Edge binaries that are fine-tuned and tested for WebView2 apps", in Microsoft's words. It comes in two distribution modes. In the **evergreen** mode the runtime is installed once on the machine and "updated automatically"; Microsoft states it receives the same updates listed in the Edge stable and security release notes. Microsoft says the evergreen runtime is preinstalled on all Windows 11 devices and was installed on eligible Windows 10 devices. In the **fixed-version** mode you package a specific runtime with your app, and Microsoft is plain about the result: it "isn't automatically updated on clients". The fixed-version binaries are over 250 MB per app.

Three details in Microsoft's documentation change fleet decisions, and they are easy to miss.

Disk is shared; processes are grouped by user data folder. All evergreen apps share one copy of the runtime on disk. On eligible systems Microsoft hard-links the Edge browser and evergreen runtime binaries when they are on the same version, which it says helps "disk footprint, memory, and performance". Processes are another matter. A WebView2 **process group** is one browser process plus renderer, GPU and other helper processes, and it is tied to one **user data folder**. Controls that use the same folder share the browser and GPU processes; an app that uses several folders gets a process group for each. Two different apps normally use different folders, so on a branch PC each app pays for its own group. Shared code pages help; private memory per app does not go away.

Installed is not running. Microsoft's distribution guide says the client uses a new runtime version "when your WebView2 app is restarted. However, if your app runs continuously, your app will continue to use the previous version." To pick up the fix, the app must release all references to the old environment objects or restart; Microsoft suggests handling the `NewBrowserVersionAvailable` event. A teller desktop that is opened at 08:00 and closed at 18:00 picks up a fix within a day. A check-in kiosk that never restarts can run an unpatched engine for weeks while every inventory tool reports the patched version installed.

Evergreen can be stopped. The same guide notes that runtime updates "might not occur if an IT Admin disables updating, or if the client is offline". Evergreen is a fleet policy, not a property of the file.

Bundled: Electron, CEF and Qt WebEngine

Bundling buys control of the version and pays for it with the patch work. The three common bundles differ mostly in how long a given engine version stays supported.

The upstream clock. Chromium announced in March 2021 that it would ship a new milestone every 4 weeks, starting with Chrome 94, and added an Extended Stable option with milestones every 8 weeks for enterprises and embedders. Every bundle below is downstream of that clock.

Electron ships a new major version every 8 weeks, and its documentation says "the latest three stable major versions are supported by the Electron team", with only the latest minor release of each line supported. At that cadence, a major version leaves support when the third newer major ships, roughly 24 weeks after its own release (3 x 8 weeks, simple arithmetic on the published policy). An app still on a major after three newer ones have shipped is out of support without anyone deciding it should be.

CEF follows Chromium's release branches. Its branch documentation (BranchesAndBuilding, checked 18 November 2026) gives the support windows in three sentences. Stable branches "receive broad security fixes for ~2 weeks after entering the stable channel." Extended branches "receive broad security fixes for ~8 weeks total (including time spent in stable)." LTC/LTS branches "receive platform-agnostic security fixes for ~9 months total (including time spent in stable and/or extended)." And: "Building legacy branches is not supported." CEF also warns that updates are manual, so there will "likely be a delay" between a Chromium release and the matching CEF build.

Qt WebEngine uses "the one used by the latest stable Chrome version at the time of Qt feature freeze", and cherry-picks security patches from newer Chrome releases into every Qt patch release. Qt says that if Chrome ships critical fixes outside its release window, "the next patch release is sped up". Qt WebEngine supports Windows, Linux and macOS, which makes it the usual Chromium option on Linux signage and embedded devices.

The **native bridge** is where a bundle's own risk lives. Electron adds Node.js and lets the page call into the app; WebView2, CEF and Qt add host-object and message channels. A sandboxed renderer is only as contained as the bridge allows. Our view (as of November 2026): in this class, a bridge that exposes too much to content it should not trust, or an engine left behind, is a larger risk than the engine choice itself.

OS-tied: WKWebView, WebKitGTK and Android System WebView

An OS-tied engine is patched by the platform owner on the platform's schedule. You do not ship the fix, and you cannot ship it either.

Apple. App Review Guideline 2.5.6 says apps that browse the web "must use the appropriate WebKit framework and WebKit JavaScript". Apple grants entitlements for other engines only in two regions. In the European Union (iOS 17.4 and later) the Web Browser Engine Entitlement covers browser apps, and the Embedded Browser Engine Entitlement covers apps that provide in-app browsing, which Apple defines as displaying web content that "would be accessible and work within a web browser app". In Japan (iOS 26.2 and later) the browser-app entitlement is similar, but the embedded entitlement is limited to apps from browser engine stewards. An app without one of these entitlements uses WebKit, and Halbrook's mobile app, whose web content is its own banking screens, is in that group. Tauri's documentation describes the consequence on the Mac: WebKit "is updated with the regular OS updates", and unsupported macOS versions do not receive WebKit updates. The age of your users' OS is the age of your engine.

Linux. WebKitGTK comes from the distribution's packages; Tauri tells developers to "check your distro's repositories". The distribution's security team is your patch owner, and its release cycle is your clock.

Android. Chromium's own documentation says that "since Android Lollipop, WebView has been an updatable system component", shipped as a separate package. On devices with Google Mobile Services and the Play Store, vendors "must use Google's provided WebView configuration" so that Google can deliver updates. Devices built on plain AOSP use the AOSP WebView package, and its updates come from whoever builds the image. Chrome and WebView have shared code: Chromium's native-library notes (as of version 125) list Monochrome, one combined package, for Android 7 to 9 and Trichrome, a shared library package, for Android 10. The current provider table no longer mentions either. At revision `f89ec29af8` of `android_webview/docs/webview-providers.md` (main, 18 November 2026), the only default provider listed for Android 10 and later (">= Q") is a standalone package: "Standalone WebView Google Stable **(default, preinstalled)**" on devices with Google services and "Standalone AOSP WebView **(default, preinstalled)**" on AOSP builds. So read "Trichrome from Android 10" as history, not as today's default. For Corvane's front-desk tablets, the first question is therefore not which engine, but whether the tablet has Google services at all.

Tauri is the deliberate version of this choice. Its `wry` library uses WebView2 on Windows, WKWebView on Apple platforms, WebKitGTK on Linux and the system WebView on Android. The installer is small because the engine is someone else's. So is the patch

schedule, and so is the version spread: one Tauri app runs on Chromium on Windows and WebKit on the Mac, with two engines to test.

Decision point	Windows	macOS	iOS	Linux	Android
OS-provided web runtime	WebView2 (Chromium)	WKWebView (WebKit)	WKWebView (WebKit)	WebKitGTK, from the distribution	Android System WebView (Chromium)
Who patches it	Microsoft, evergreen; you, in fixed-version mode	Apple, with macOS updates	Apple, with iOS updates	The distribution	Google through Play on devices with Google services; the image builder otherwise
When the fix is live	When the app releases the old environment or restarts	After the OS update installs	After the OS update installs	After the package update and app restart	After the package update and app restart
Bundled alternative	Electron, CEF, Qt WebEngine	Electron, CEF, Qt WebEngine	Only with an Apple entitlement: browser and in-app browsing apps in the EU; browser apps and engine stewards' apps in Japan	Electron, CEF, Qt WebEngine	Uncommon; GeckoView exists

One CVE, many copies: the libwebp week

September 2023 shows what these decisions mean in days. All dates below are UTC; where a source gives only a date, that date is used. **libwebp** is the reference decoder for the WebP image format. A heap buffer overflow in its lossless decoder, CVE-2023-4863, could be triggered by a crafted image, and it was being exploited. Every runtime in this chapter decodes images, so every runtime had to move.

- **6 September.** Apple's Security Engineering and Architecture team and the Citizen Lab reported the bug to Chrome, according to Chrome's release notes.
- **7 September.** Apple shipped iOS 16.6.1 and macOS updates for CVE-2023-41064, a buffer overflow in its own image handling, and the Citizen Lab published its report on the BLASTPASS exploit chain the same day. This is a different bug in Apple's own code, shown for context, not a libwebp fix.
- **11 September.** Chrome 116.0.5845.187 shipped, with Google stating it was "aware that an exploit for CVE-2023-4863 exists in the wild".
- **12 September.** NVD published the CVE. Microsoft shipped Edge 116.0.1938.81 with the fix (Edge security release notes, entry dated 12 September); evergreen WebView2 receives the same updates. Electron tagged 26.2.1, 25.8.1, 24.8.3 and 22.3.24 the same day

(between 15:40 and 18:47 UTC), the last three with a "backported fix for CVE-2023-4863", and published the 26.2.1 release page. Chrome 117 for Android (117.0.5938.60) shipped the same day; Google says Android releases carry the same security fixes as the matching desktop release, and desktop Chrome 117 lists CVE-2023-4863. Google posted no separate date for Android System WebView.

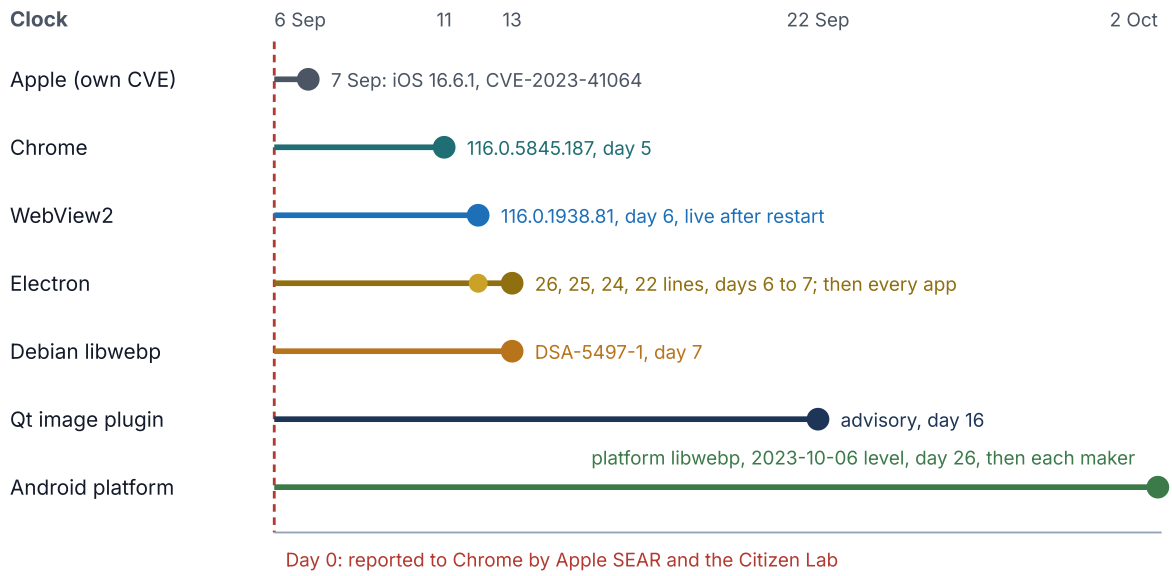
- **13 September.** The release pages for Electron 25.8.1, 24.8.3 and 22.3.24 went up. libwebp 1.3.2 was released (its release commit is dated 22:11 UTC; the tag followed at 01:49 UTC on 14 September). CISA added the CVE to its Known Exploited Vulnerabilities catalog. Debian shipped its libwebp update (DSA-5497-1, mailed 21:07 UTC).
- **15 September.** Microsoft listed the same fix for Edge Extended Stable 109.0.1518.140 and for the new Edge 117.
- **22 September.** Qt published an advisory for the WebP plugin in its `qtimageformats` module, which is not Qt WebEngine: fixed in Qt 5.15.16, 6.2.10, 6.5.3 and 6.6.0. Qt WebEngine bundles its own Chromium copy and takes such fixes through its patch releases; we found no separate Qt WebEngine advisory for this CVE. The 22 September date belongs to a different copy of libwebp, not to an embedded runtime.
- **2 October.** The Android Security Bulletin listed CVE-2023-4863 as under "limited, targeted exploitation". The fix is in the System component, Android's own platform copy of libwebp, at the 2023-10-06 patch level, which each device maker ships on its own schedule. The bulletin does not date Android System WebView, which carries its own Chromium copy and updates through Google Play.

Read those dates by component. For the runtimes in this chapter, the comparable clocks are Chrome (day 5), Edge and evergreen WebView2 (day 6) and Electron (days 6 to 7), with Debian's system libwebp, which WebKitGTK uses there, at day 7. The Qt plugin advisory and the Android bulletin date other copies of the same library, and Apple's fix is a different bug: they show how far one flaw spreads, not how fast one runtime moves. We found no published fix date for Qt WebEngine or for Android System WebView.

Those are the vendors' dates. Each of them is the start of a second clock: Electron's release does nothing until each app that bundles Electron ships an update and its users install it, and the WebView2 fix does nothing on a kiosk until the app restarts. The ledger column "Who patches, and how fast" exists to hold both numbers.

FIGURE 11.1

One CVE, many copies: CVE-2023-4863 in September 2023



Each dot is the vendor's release. Your devices start a second clock at that dot.

Sources: Chrome Releases (11 Sept 2023), Microsoft Edge security release notes, Electron release pages, NVD and CISA KEV, Debian DSA-5497-1, Qt advisory (22 Sept 2023), Android Security Bulletin October 2023. Apple's 7 September fix is a separate CVE in its own image code, shown for context. The Qt row is the image plugin, not Qt WebEngine, and the Android row is the platform library, not Android System WebView; only the Chrome, WebView2, Electron and Debian rows date an embedded runtime's engine or its system library.

> *An embedded runtime is a contract with an update clock. Read the clock before you read the feature list.*

HOW IT GOT THIS WAY

Embedding a browser engine is older than most of today's runtimes. CEF's repository begins with a commit on 26 November 2008, weeks after Chrome's first release. GitHub built its Atom editor on a shell it called Atom Shell; the project was renamed Electron in 2015 and reached 1.0 on 11 May 2016. Android made WebView an updatable package with Lollipop in 2014. On 6 December 2018 Microsoft said it intended "to adopt the Chromium open source project in the development of Microsoft Edge on the desktop"; the new Edge shipped on 15 January 2020, and WebView2 for Win32 apps reached general availability on 19 October 2020, with fixed-version mode and .NET following on 20 November 2020. That left three engine families, Blink, WebKit and Gecko, under almost every embedded runtime, and made the update clock, not the engine, the real choice.

What hasn't changed: every embedded runtime is still a full browser engine with its own update clock, and in 2023 one flaw in one image decoder still needed a separate release from every owner of a copy of it: from day 5 for Chrome to day 26 for Android's platform library in Figure 11.1. Checked 17 November 2026.

THE UNSEEN ANGLE

The runtime version is a liability schedule, and the hidden line in it is **activation**. Microsoft documents that an evergreen fix reaches a running WebView2 app only when the app releases its environment or restarts. Inventory tools report what is installed. For a kiosk or a teller desktop that runs for days, the number that matters is time to *running* the fix. Put an activation SLA, not only an install SLA, into the runtime section of every contract and every kiosk image, and test it by checking the version the running process reports. Source: Microsoft Learn, "Distribute your app and the WebView2 Runtime", section "Handle Evergreen WebView2 Runtime updates".

This hop in the Frame Ledger

The worked example is Halbrook Bank's payment-approval screen on 2,500 branch PCs, in a Windows desktop app that embeds WebView2 in evergreen mode.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
1 to 3 Parse, style, layout: Blink in WebView2, evergreen	Chromium upstream; Microsoft ships and patches	Microsoft, on the Edge security cadence; live only after the teller app restarts	Renderer, sandboxed	The page, the native bridge, UI Automation	**RENT**, **OWN**: the restart policy (daily restart, plus `NewBrowserVersionAvailable` handler)
4 and 6 Text and image decode: font stack and libwebp inside the runtime	Upstream libraries; Microsoft ships	As above; in 2023, six days from report (6 September) to Edge release (12 September)	Renderer, sandboxed	Anyone who can get an image or font onto the screen	**ISOLATE**, **RENT**: no remote images or fonts on the approval screen
0 Source and host: native bridge, host objects and web messages	Halbrook's own code	Halbrook, on its release train	App process, not sandboxed	Any script the page runs	**OWN**: allow-list messages, load only Halbrook origins
7 Engine compositor: Chromium cc and viz	Chromium upstream; Microsoft ships	With the runtime	GPU process, shared by controls on the same user data folder	Not app-readable	**RENT**: keep the GPU sandbox on, no debug flags in production
1 to 7 Runtime activation: the whole engine	Halbrook end-user computing	Target: fix running within 24 hours on every branch PC	Whole process group	Fleet tooling, if it reads the running version	**OWN**, **PIN**: own the SLA; pin only through Microsoft's update policies, never by disabling updates

Partial rows for the other runtimes, set on the companies that would use them:

Runtime and screen	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision
Electron, Larkfield's main window	Electron project; Larkfield ships	Larkfield, after each Electron release; supported for about 24 weeks per major	Chromium processes plus Node in the main process	Page, preload bridge, OS accessibility	**OWN**, **ISOLATE**: own the upgrade train; context isolation and sandboxed renderers
WebView2, Corvane check-in kiosk	Microsoft; Corvane runs it	Microsoft; live only after the kiosk app restarts	Process group per user data folder	Guest, capture tools, kiosk agent	**RENT**, **OWN**: a nightly restart window
Android System WebView, Corvane front-desk tablet	Google on devices with Google services; the image builder otherwise	Play updates, or the device maker's images	Renderer process per app	Staff, AccessibilityService apps	**RENT**: buy only tablets with Google services or a written WebView update term
Qt WebEngine, Corvane Linux signage	Qt; Corvane's integrator ships	Qt patch releases with cherry-picked Chrome fixes, then the integrator	Chromium process model	Anyone in the lobby	**PIN**, **OWN**: pin to a Qt patch line; own the rebuild
WKWebView via Tauri, a Larkfield macOS pilot	Apple	Apple, with macOS updates; none on unsupported macOS	WebKit content process	Page, bridge, macOS accessibility	**RENT**: set a minimum supported macOS

SECURITY LENS

Who can change it? In evergreen mode Microsoft changes the engine under your app, and IT can stop it. In bundled mode only you can change it, and only by shipping a release. In OS-tied mode the platform owner changes it on the platform's schedule. In every mode, your team owns the native bridge.

How fast can it be fixed? In September 2023 the Chromium-based runtimes shipped in five to seven days (Chrome, WebView2, Electron), counted from the 6 September report. Other copies of libwebp took longer, up to 26 days for Android's platform bulletin, and Google published no separate date for Android System WebView. Add your own clock: the restart for evergreen, your release and your users' installs for bundled, the OS update for OS-tied.

What does AI change here? Our view (as of November 2026): AI-assisted fuzzing and exploit work shorten the time between a public fix and a working exploit, so the second clock, from vendor release to running fix on your devices, is the one to shrink. Agents that drive embedded runtimes through automation ports or the accessibility tree also add a new caller to the native bridge; treat agent access as a separate, logged path.

BUSINESS IMPLICATIONS

Cost. Bundling moves the patch bill to you: Larkfield pays for an Electron upgrade at least every 24 weeks on 400,000 desktops, plus the memory of one engine per app on each of them. Evergreen moves it to Microsoft, but Halbrook still pays for restart windows and forward-compatibility testing against Edge preview channels. Corvane pays most for a mixed fleet: three engines, three owners, three clocks.

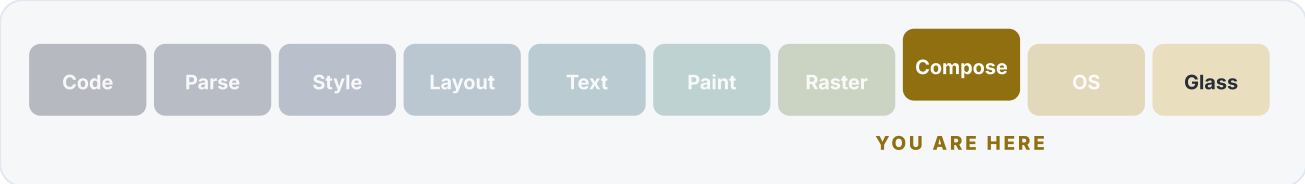
Visibility. Inventory tools usually report which engine versions are installed, not which are running, across your apps. Halbrook can read the running version from the runtime and report it; Larkfield can report its bundled Chromium version from its own telemetry. Make both part of the fleet dashboard.

Value. One web codebase across Windows, macOS and Linux is real value, and it is why Larkfield chose Electron. Price it honestly: the saving in UI engineering is partly spent again on patching and memory, and that trade belongs in the business case, not in the incident review.

Monday rule: for every app that embeds a browser engine, write down its decision (evergreen, bundled, OS-tied), its patch owner, and the time from vendor release to fix running on your devices.

CHAPTER TWELVE

The Native Path: Composition, Skia and the Rust Renderers



The question "Electron or native?" hides the decision that matters: which hops of the frame you are willing to own. Leaving the browser engine does not leave rendering; it hands you a scene graph, a raster library, a GPU interface and, most expensive of all, the accessibility tree that the browser used to build for free. This chapter separates the framework from the rendering layer, shows what each native stack composes with, and gives a CTO a test for when owning those hops pays back.

The framework is not the renderer

A UI **framework** is what your developers write against: WinUI, SwiftUI, Jetpack Compose, GTK, Qt, Flutter. Underneath it sits a **rendering layer**: a tree of visual objects that is kept between frames and handed to a compositor, plus a library that turns shapes and text into pixels. The framework decides how pleasant the code is. The rendering layer decides frame cost, memory and which owner you depend on for fixes.

The mainstream native stacks share one model, the **retained scene graph**: the app describes layers or nodes once, changes them when state changes, and a compositor draws the result, often off the UI thread.

- **Windows.** Microsoft describes composition visuals as "the visual tree structure that all other features of the Microsoft.UI.Composition API use and build on", and WinUI 3 apps get their compositor from that tree. Its ancestor, DirectComposition, "was introduced in Windows 8" and operates "independently of the UI thread".
- **macOS and iOS.** Every UIKit view is backed by a Core Animation layer. On macOS, windows of apps linked against the 10.14 SDK or later are displayed using Core Animation, but Apple notes this "doesn't mean that all views are layer-backed": a view has its own layer or draws into a layer shared with others. SwiftUI makes no promise of one layer per view. On macOS, WindowServer composes the result into the screen; on iOS, Core Animation's render server, the `backboardd` process, does that job.

- **Linux.** GTK 4, released in December 2020, renders through GSK, its scene graph kit. Qt Quick uses a scene graph that is "traversed and rendered via a graphics API" such as Vulkan, Metal or Direct3D, and on many platforms is "rendered on a dedicated render thread".
- **Android.** Views and Jetpack Compose (1.0 on 28 July 2021) draw through HWUI, Android's hardware-accelerated UI renderer, which is built on Skia; skia.org lists Android among the products Skia draws for.

None of this removes the OS compositor from Chapter Five. Native apps can reach the glass with fewer copies, through overlays or direct flip, but on a composited desktop the OS compositor still decides when the frame is shown.

Toolkits that draw every pixel themselves

A second family ignores the platform's widgets and paints everything through its own renderer. That gives one look on every OS and one code path to profile. It also makes the toolkit, not the OS, responsible for everything the OS widgets used to do.

Skia is the shared 2D library under much of this. Its own site says Skia is "sponsored and managed by Google" and "serves as the graphics engine for Google Chrome and ChromeOS, Android, Flutter, and many other products". On 8 July 2025 Google announced Graphite, a new Skia rasterization backend for "modern graphics APIs like Metal, Vulkan and D3D12", launching first in Chrome on Apple Silicon Macs.

Flutter draws its widgets itself. It began on Skia and replaced it with its own renderer, Impeller: Flutter 3.10 (May 2023) made Impeller the default on iOS, and Flutter 3.27 (December 2024) made it the default on Android API 29 and later, with "the Skia rendering engine" still used on older devices and devices without Vulkan. Flutter 3.29 (February 2025) changed that fallback: devices without a working Vulkan driver "will fall back to Impeller running on OpenGLES instead of using Skia". Flutter 3.47 (August 2026) made Impeller the default on macOS, Windows and Linux, on "Metal on macOS and Vulkan on Windows and Linux". Flutter documents no Direct3D backend for Impeller.

The Rust renderers are components, not yet whole platforms.

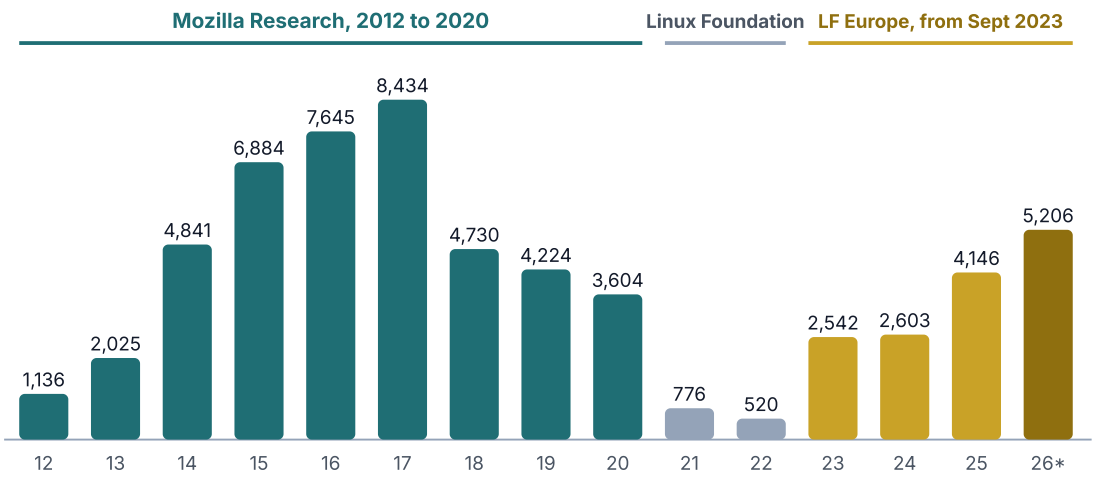
- **wgpu** is "a cross-platform, safe, pure-Rust graphics API" that runs on Vulkan, Metal, D3D12 and OpenGL and "serves as the core of the WebGPU integration in Firefox, Servo, and Deno". Deno shipped it first, in Deno 1.8 (March 2021); Firefox enabled WebGPU on Windows in Firefox 141 (July 2025). Chrome uses its own WebGPU implementation, Dawn, and Safari uses its own stack, so wgpu is not under every browser. Its use in Firefox is still the strongest production evidence a Rust graphics layer has.

- **Vello** is "a GPU compute-centric 2D renderer" that "performs most rendering work in GPU compute shaders". Its own README calls the compute renderer "an experimental implementation" and says its CPU variant is "currently overall more mature".
- **Servo** is a whole web engine in Rust, independent and embeddable. Its first release, v0.0.1, was published with binaries on 20 October 2025 (the tagged commit is dated 19 October), and v0.8.0 on 9 November 2026 (tagged commit 5 November). Since v0.1.0 (April 2026) the release tags match the version of the `servo` crate on crates.io (0.8.0 published 7 November 2026), with the 0.1.x line marked long-term support: active, and still before 1.0. Figure 12.1 holds this book's one set of Servo commit measurements; Chapter Sixteen and Appendix D refer back to it.

Our view (as of November 2026): rent Rust renderers where they are proven, as wgpu is under WebGPU in shipping browsers, and treat a whole Rust engine as a component to pilot behind a feature flag, not a platform to bet a product on.

FIGURE 12.1

Servo commits per year, measured from the repository



Year (20xx). *2026 to 18 November. Commits on the main branch by commit date; activity, not funding or headcount.

Measured: `git rev-list --count --since=YYYY-01-01 --until=YYYY-12-31 HEAD` in the Servo repository (github.com/servo/servo), head `ff0b018707`, cloned 18 November 2026. First commit `ce30d4520d`, 8 February 2012, "initial add"; 59,387 commits in total.

What you take on: accessibility, text and input

A browser engine and the platform widget sets do three expensive things that demos never show: they publish an accessibility tree, they lay out and shape text in every script, and they integrate with input methods. A stack that draws its own pixels inherits all three as work.

Accessibility. Screen readers, voice control and test tools read the accessibility tree, not the pixels; AI agents read it too, alongside screenshots and the DOM (Chapter Ten). A custom renderer must expose that tree itself. Android's Compose documentation is direct about it: the semantics tree is filled automatically for standard components, "however when you're adding custom low-level composables, you have to manually provide its semantics". Its example is a calendar drawn straight to a canvas, which TalkBack can only announce as "Calendar" until the developer adds a node for each day. Flutter keeps a separate semantics tree of `SemanticsNode` objects for the same reason. In Rust, AccessKit exists to make it "easier to implement accessibility, for screen readers and other assistive technologies, in toolkits that render their own user interface elements", and maps one tree to UI Automation on Windows, NSAccessibility and UIAccessibility on Apple platforms, AT-SPI on Linux and Android's accessibility API.

Text. Shaping, bidirectional text, font fallback and line breaking (Chapter Three) come with the browser and with platform text views. A toolkit that draws its own text must bring a shaping and layout stack, and its results in Arabic, Thai or Chinese are only as good as that stack.

Input methods. Typing Chinese, Japanese or Korean goes through an input method editor that needs to know where the caret is and what text surrounds it. Platform text controls answer those questions. A custom text field has to answer them too, on each OS's API.

None of this argues against native. It argues for pricing it. **Our view (as of November 2026):** the accessibility and text work, not the drawing, is where the estimate for a custom renderer usually goes wrong.

Layer you own on the native path	Windows	macOS	iOS	Linux	Android
Native UI and composition layer	WinUI 3 on Microsoft.UI.Composition; Win32 on DWM	AppKit, SwiftUI on Core Animation	UIKit, SwiftUI on Core Animation	GTK 4 on GSK; Qt Quick scene graph	Views and Compose on HWUI and Skia
GPU interface under a custom renderer	Direct3D 12 (wgpu); Direct3D 11 through ANGLE (Skia in Chromium); Vulkan (Impeller, from Flutter 3.47)	Metal	Metal	Vulkan, OpenGL	Vulkan, OpenGL ES
Accessibility API a custom renderer must feed	UI Automation	NSAccessibility	UIAccessibility	AT-SPI	Android accessibility API

When native pays back

Native is not one decision. A team can own one hot view and rent the rest. Four tests decide it.

- **Latency you can measure.** If the screen misses frames in a trace and the misses sit in the web engine's layout or paint, a native view on the platform compositor removes a whole layer. If the misses sit in your data path, native will not help.
- **Memory per seat.** One native window does not carry a browser engine's process group. On thin clients and VDI that difference is a hardware line, not a preference.
- **Patch ownership.** A native app on a toolkit that ships with the OS, such as Win32, AppKit or UIKit, is patched with the OS. WinUI 3 is not: the Windows App SDK is serviced through its own framework package, or, in a self-contained app, only when the publisher ships a new release. A custom renderer is patched by you, plus the GPU drivers it talks to. Memory-safe Rust removes classes of bugs in your own code; it does not remove driver bugs.
- **Accessibility budget.** If the view must pass an accessibility audit and you cannot fund building and testing its tree on every OS, keep it on platform widgets or in the web engine.

A mixed answer is common and often right. Larkfield can keep its Electron shell for settings, sign-in and help pages, and draw only the analytics canvas natively, where a trace shows the time is spent. Corvane can keep web content on its signage and move the check-in flow, the one screen every guest touches, to a native view on fixed kiosk hardware. In both cases the Frame Ledger records two sets of rows for one app: the rented engine's, and the owned renderer's, with a different patch owner for each.

> *Leaving the browser engine does not leave rendering. It hands you the scene graph, the raster and the accessibility tree, and the last of those is the one nobody budgets.*

HOW IT GOT THIS WAY

The retained scene graph arrived on the desktop with Core Animation, "first introduced in OS X v10.5", Leopard, in 2007, and on Windows with DirectComposition in Windows 8 in 2012. Mozilla Research created Servo in 2012 as "the first major Rust codebase other than the compiler itself". Parts of it went to Firefox: Mozilla credits Servo's parallel styling engine, Stylo, for much of Project Quantum in 2017, and WebRender for later rendering gains. On 11 August 2020 Mozilla announced it would cut about 250 roles; on 17 November 2020 the Linux Foundation announced it would host Servo; on 7 September 2023 Servo joined Linux Foundation Europe, with renewed work led by Igalia. Figure 12.1 shows those years in the commit record. The decision today follows from that history: Rust renderers are real components to rent, and an engine's future depends on who funds its team.

What hasn't changed: a retained tree of visuals handed to a compositor, the model of Core Animation (2007) and DirectComposition (2012), is still how WinUI, SwiftUI, GTK 4, Qt Quick and Compose draw. The Servo measurements behind this history are in Figure 12.1 and Appendix D.

THE UNSEEN ANGLE

A custom renderer does not remove accessibility; it moves the work to you. The cost shows up in three places a demo never visits: a screen reader announcing "Calendar" instead of the selected day, an input method that cannot find the caret, and an agent working from the accessibility tree that finds a blank canvas where your buttons are. Ask any vendor or team proposing a custom-drawn UI for its semantics tree on each OS, and test it with a screen reader before you test the frame rate. Sources: Android Developers, "Semantics in Compose"; AccessKit README.

This hop in the Frame Ledger

Larkfield Software is piloting a native analytics canvas, drawn with Vello on wgpu, inside its Electron app's hottest view on Windows and macOS.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
0 to 3 Source to layout: scene and layout of the canvas view	Larkfield	Larkfield, on its release train	App process, no browser sandbox	Only through what Larkfield exposes	**OWN**: Larkfield's code, on its own release train
6 Raster: Vello, open-source crates; its GPU compute renderer is experimental by its own README	Linebender community; Larkfield pays in engineer time	Upstream when it ships; Larkfield when it upgrades crates	App process	Whatever it draws	**PIN** a tested version; pilot only, and benchmark Vello's CPU path beside the GPU path; fund upstream fixes you need
6 to 7 GPU interface: wgpu over Direct3D 12 and Metal, then the GPU driver	gfx-rs community; GPU vendors and Apple for drivers	wgpu on upgrade; drivers through Windows Update, vendor tools, macOS updates	App process and kernel driver	Capture APIs via the OS compositor	**RENT** wgpu; **PIN** tested driver versions on managed fleets
10 Glass and readers: accessibility tree via AccessKit to UI Automation and NSAccessibility	Larkfield builds the nodes	Larkfield	App process	Screen readers, test tools, AI agents	**OWN**: audit on each OS before release
8 OS compositor: DWM, WindowServer	Microsoft, Apple	With the OS	System process	User, capture APIs	**RENT**: patched with the OS

SECURITY LENS

Who can change it? On platform toolkits, the OS vendor. On a custom renderer, your team and the open-source maintainers of every crate or library you pin, plus the GPU driver vendors underneath. The browser's sandbox no longer surrounds your drawing code unless you build a process boundary yourself.

How fast can it be fixed? As fast as you ship. That is a strength when you own the bug and a weakness when the bug is upstream and nobody on your team watches that project's advisories.

What does AI change here? Agents that work from the accessibility tree find nothing in a custom renderer without one; agents that read screenshots still see the pixels, but act with no semantics to check them against. A careless tree can steer tree-based agents with labels that disagree with the pixels. Treat the semantics tree as a security surface: the amount and payee in the tree must match what is drawn.

BUSINESS IMPLICATIONS

Cost. Larkfield should price the native canvas as three budgets: drawing, accessibility and text input. On 400,000 desktops the memory saved per seat is likely, but Larkfield should measure it on a reference desktop before it counts it; the cost of supporting two accessibility APIs, UI Automation and NSAccessibility, is certain, plus one more for each OS the view later reaches. Corvane's kiosks and signage gain most from native views, because they run one screen for years on fixed hardware.

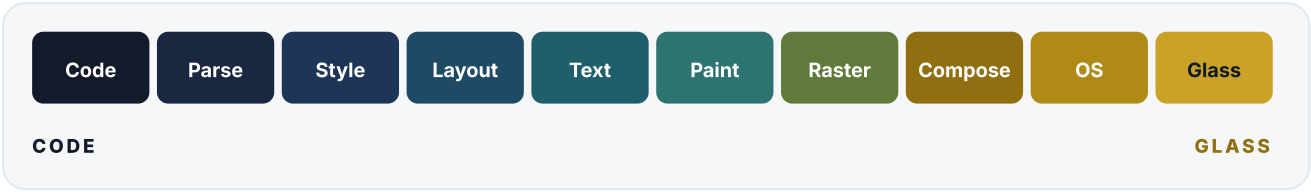
Visibility. Owning the renderer means owning the trace. Larkfield can time every hop in its own code, which the web engine never allowed; Halbrook gains the same on any native teller view, and loses the browser's built-in developer tools.

Value. Owning the compositor boundary is how a team owns latency on its most used screen. Rent the rest.

Monday rule: before approving any custom-drawn UI, ask for its accessibility tree on each target OS and a screen-reader test of the most critical screen.

CHAPTER THIRTEEN

Who Pays for the Engine: The Money Story



The engine that draws your customer's screen costs you nothing to download, and someone pays for it every year. That payer sets the engine's priorities, its patch capacity and, in the end, how long it exists. This chapter changes one decision: whether your rendering roadmap treats the engine as a free utility, or as a dependency funded by someone else's business, with records you can read. It uses only those records: court opinions, audited statements, vendor statements and the engines' own commit history.

Who writes the engines: the commit record

Three engine families have drawn most web content since January 2020, when Microsoft's new Edge replaced its own EdgeHTML engine with Chromium: Blink (in Chromium), WebKit and Gecko. Microsoft ended support for the EdgeHTML-based Edge Legacy on 9 March 2021. Its older MSHTML (Trident) engine survives only inside Internet Explorer mode in Edge, which Microsoft supports through at least 2029 (Chapter Fifteen). Their repositories are public, so one question can be answered from the source itself: whose email domains are on the commits?

A count of every non-merge commit made in calendar year 2025 on the main branch of each repository, grouped by the domain of the author's email, with automated service accounts removed from the Chromium count, gives Figure 13.1. Read it with care. A commit count measures activity, not money or effort; a large change and a typo fix count the same. Domains are a proxy too: many Google engineers commit from `chromium.org` addresses, but so do others, and a `gmail.com` address says nothing about an employer.

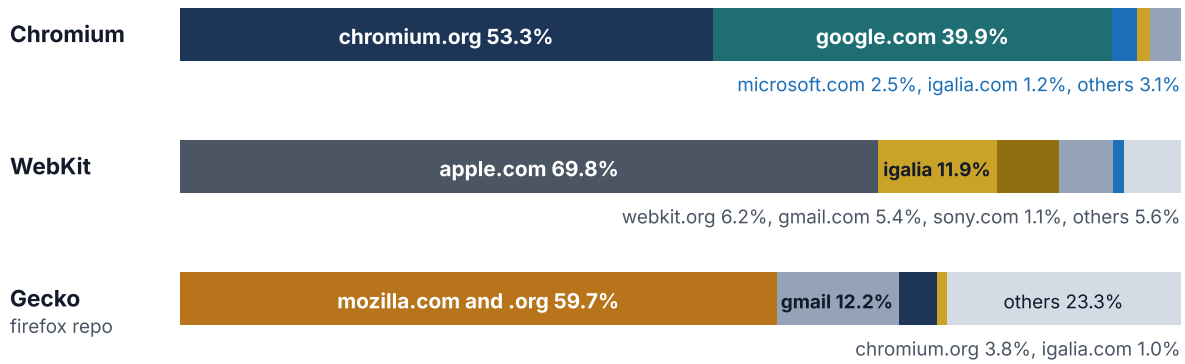
- **Chromium** (109,790 commits): `chromium.org` 53.3 percent, `google.com` 39.9, `microsoft.com` 2.5, `igalia.com` 1.2, `intel.com` 0.5, `opera.com` 0.4, all other domains 2.2.
- **WebKit** (16,672 commits): `apple.com` 69.8 percent, `igalia.com` 11.9, individual `webkit.org` addresses 6.2, `gmail.com` 5.4, `sony.com` 1.1, all other domains 5.6.

- **Gecko**, measured in the firefox monorepo (`mozilla-firefox/firefox`, which holds Gecko together with the Firefox front end; 54,305 commits): `mozilla.com` and `mozilla.org` together 59.7 percent, `gmail.com` 12.2, `chromium.org` 3.8 (all 2,048 of these commits carry "[wpt PR" in the subject line, which marks web-platform-tests syncs, shared test work imported from upstream), `igalia.com` 1.0, all other domains 23.3.

The pattern is plain even with the caveats. Each engine is written mostly by one company: Google for Chromium, Apple for WebKit, Mozilla for Gecko. Microsoft is the largest named contributor to Chromium outside Google. Igalia, a consultancy that is paid by other companies to do engine work, is the second-largest domain in WebKit and appears in all three.

FIGURE 13.1

Who commits to the three engine families, 2025



Share of non-merge commits on the main branch, 1 Jan to 31 Dec 2025, by author email domain. Activity, not funding.

Measured 18 November 2026 with `git log --no-merges --since=2025-01-01 --until=2025-12-31 --format=%ae` on chromium/src (head `e52d393f6d`), WebKit (head `f31b755f63d`) and mozilla-firefox/firefox (head `524e4e57a3`). Chromium count excludes `gserviceaccount.com` and `appspot` service accounts.

The money behind the defaults: US v. Google

The most detailed adjudicated public record of default-search economics comes from a federal court. In *United States v. Google LLC*, No. 1:20-cv-03010 (D.D.C.), Judge Amit Mehta's memorandum opinion of 5 August 2024 concluded: "Google is a monopolist, and it has acted as one to maintain its monopoly. It has violated Section 2 of the Sherman Act." The findings of fact put numbers on the contracts that set default search engines in browsers and devices.

- "In 2021, Google paid out a total of $26.3 billion in revenue share under these contracts", the agreements with Apple, Mozilla, Android device makers and US wireless carriers (finding 289).
- "In 2022, Google's revenue share payment to Apple was an estimated $20 billion" (finding 299).
- "Google's 2021 revenue share payment to Mozilla was over $400 million, or about 80% of Mozilla's operating budget" (finding 335). The court added that "Mozilla has repeatedly made clear that without these payments, it would not be able to function as it does today."
- On Chrome itself, the court recorded that "Chromium is the underlying engine which powers Chrome", that it is open source, and that "Google is the default search engine on Chrome" (finding 6).

The court's record says nothing about engine budgets, and this book does not guess at them. What it does establish is the flow: search revenue pays for default placement in browsers, and two of the three engine families sit inside browsers that receive those payments.

The remedies, and what they kept

The remedies opinion of 2 September 2025 (ECF No. 1436) set the terms that now shape that flow. Its top-line determinations:

- Google "will be barred from entering or maintaining any exclusive contract relating to the distribution of Google Search, Chrome, Google Assistant, and the Gemini app", including contracts that tie revenue share to keeping those products on a device or access point "for more than one year".
- Google "will not be required to divest Chrome", and the judgment includes no contingent divestiture of Android.
- Google "will not be barred from making payments or offering other consideration to distribution partners for preloading or placement". The court reasoned that cutting off payments "almost certainly will impose substantial" downstream harms, "in some cases, crippling", on "distribution partners, related markets, and consumers".
- Google must make certain search index and user-interaction data, and search and search text ads syndication, available to qualified competitors.

The opinion records why the payment question mattered to engines. Mozilla's witness testified that lower revenue share could "potentially start a downward spiral of usage as people defected from our browser, which . . . could at the end of the day put Firefox out of business". The payments continue; exclusivity does not.

Mozilla's audited statements

Mozilla publishes consolidated financial statements with an independent auditors' report. For the year ended 31 December 2024:

- royalties were $498.2 million of $680.4 million in total revenue and support without donor restrictions, about 73 percent (computed from the statement);
- the notes describe royalties as a percentage of revenue earned by search partners whose engines are placed in Firefox;
- "Approximately 86% and 85% of Mozilla's revenues from customers with contracts were derived from one customer" in 2024 and 2023;
- software development expense was $290.4 million.

The statements do not name the customer. Read with finding 335 of the 2024 opinion, the dependence is on the record, and it has not changed shape since 2021.

Platform owners: Microsoft's switch and Apple's rule

Two platform decisions shaped who pays for the engine under ordinary apps.

Microsoft. On 6 December 2018 Microsoft said it intended "to adopt the Chromium open source project in the development of Microsoft Edge on the desktop", citing "better web compatibility for our customers and less fragmentation of the web for all web developers". Since the new Edge shipped on 15 January 2020, Microsoft has funded its browser and its app runtime, Microsoft's Edge WebView2 runtime, by contributing to an upstream that Google mostly writes (Figure 13.1), and by shipping and patching the result itself.

Apple. App Review Guideline 2.5.6 says apps that browse the web "must use the appropriate WebKit framework and WebKit JavaScript". In the European Union, from iOS 17.4, Apple offers a Web Browser Engine Entitlement for browser apps and an Embedded Browser Engine Entitlement for apps that provide in-app browsing, defined as "the display of content dynamically from the web that would be accessible and work within a web browser app"; both require distribution solely in the EU. In Japan, from iOS 26.2, the embedded entitlement is limited to apps from browser engine stewards. Everywhere else, and for apps without an entitlement, Apple's WebKit is the engine, and Apple alone pays for and patches it.

The libraries and runtimes. Skia is "sponsored and managed by Google". Electron entered the OpenJS Foundation's incubation program in December 2019 and became an OpenJS Impact Project on 23 June 2020; the foundation hosts the project, and the engine inside it is Chromium. Servo is hosted by Linux Foundation Europe, with work led by Igalia since 2023 (Chapter Twelve).

Reading the money as roadmap risk

None of these records tells you what a vendor will do next, and this book does not guess. They do tell you where to look, and three signals are worth a line in any supplier review.

- **Concentration.** When one customer supplies about 86 percent of an engine owner's contract revenue, a change in that one contract is a change in the engine's budget. Watch the counterparty's court filings and the owner's next audited statement.
- **Downstream distance.** Microsoft, every Electron and CEF vendor, and Qt all ship an engine they mostly do not write. Their fixes arrive after upstream's, so the upstream's capacity sets the floor on your patch time, and the downstream's release process adds to it.
- **Platform rules.** Apple's engine rule and its EU and Japan exceptions decide which engine your iOS screens can use. A rule change can open or close an option for your mobile roadmap without any change in your code.

A team that records these three signals per engine turns "the browser is free" into a supplier risk it can review once a year, like any other critical dependency.

Default engine and its payer	Windows	macOS	iOS	Linux	Android
Engine under the OS web runtime	Chromium (WebView2)	WebKit	WebKit	No OS web runtime: WebKitGTK, Qt WebEngine or a bundled Chromium	Chromium (System WebView)
Who mostly writes it (2025 commits)	Google upstream; Microsoft ships	Apple	Apple	Apple writes most of WebCore; Igalia maintains the WebKitGTK port; the distribution packages it	Google
Platform rule	Free choice	Free choice	WebKit, except entitled apps in the EU and Japan	Free choice	Free choice

> *Your rendering roadmap is funded by someone else's advertising or platform strategy. Read their records the way you read a supplier's accounts.*

HOW IT GOT THIS WAY

Building a web engine has become a job only a few organizations fund. Microsoft announced on 6 December 2018 that it would build Edge on Chromium, and the new Edge shipped on 15 January 2020, leaving three engine families. The 2024 opinion documents the payment flow that sustains browsers: $26.3 billion in revenue share in 2021 alone. The 2025 remedies ended exclusive defaults and kept the payments, so the flow that funds two of the three families continues under new terms. Decision today: treat each engine as a supplier whose income you can read, and whose income can change by court order or platform policy.

What hasn't changed: three engine families, each written mostly by one company, as measured in Figure 13.1 for 2025. Measured 18 November 2026 on chromium/src `e52d393f6d`, WebKit `f31b755f63d` and firefox `524e4e57a3`.

THE UNSEEN ANGLE

The commit record shows who writes the engine under your app, year by year, from the source itself rather than from marketing. Two findings are easy to miss. Microsoft, which ships the engine under every evergreen WebView2 app, wrote about 2.5 percent of Chromium's 2025 commits by this measure, so its patch speed depends on an upstream it does not control. And Igalia, a contractor, wrote about 12 percent of WebKit's, which means outside companies pay to move the engine that Apple controls. Run the same three-line `git log` on any engine you depend on before you sign a multi-year roadmap on it. Source: Figure 13.1 and `sources/ch13.md`.

This hop in the Frame Ledger

Halbrook Bank fills the "Owner and who pays" column honestly for its three customer-facing engines.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
1 to 7 Engine: Blink and Skia in WebView2, teller desktop	Written mostly by Google, funded by its business; Microsoft ships and pays for its own build	Microsoft, on the Edge security cadence, after upstream fixes	Renderer and GPU processes, sandboxed	Page, bridge, UI Automation	**RENT**: track Microsoft's security release notes, not only Chrome's
1 to 7 Engine: WebKit, iOS app web screens	Apple, funded by its platform	Apple, with iOS updates	WebKit content process	App, accessibility clients	**RENT** and **PIN** a minimum iOS version
1 to 7 Engine: System WebView, Android app web screens	Google	Google through Play on devices with Google services	Renderer process	App, AccessibilityService	**RENT**: check the WebView version at run time
1 to 7 Engine: Gecko, in advisor laptops' Firefox	Mozilla, about 73 percent funded by royalties in 2024	Mozilla release cadence	Gecko content processes	User, extensions	**RENT**: record the funding dependency in the supplier risk file

SECURITY LENS

Who can change it? The company that writes most of the commits sets the engine's direction; downstream shippers such as Microsoft and every Electron vendor follow it. Courts and regulators can now change the money that pays for that work, as the 2025 remedies and the EU and Japan rules show.

How fast can it be fixed? Patch capacity follows funding. Our hypothesis is that an engine whose main payer cuts back loses fixers before it loses features. Commit counts cannot test that, because they measure activity, not who left or what they worked on; Servo's 2021 and 2022 counts (Figure 12.1) show only that activity fell when funding did.

What does AI change here? AI-assisted bug finding raises the number of reports every engine team must triage, so patch capacity, and the money behind it, matters more each year. AI products are also now named in the remedies: Google may not make the Gemini app part of exclusive distribution deals, which ties future assistant placement to the same rules as search.

BUSINESS IMPLICATIONS

Cost. The engine is free to Halbrook, Larkfield and Corvane because users' searches and Apple's platform pay for it. If those flows shrink, costs return as slower patches or fewer features, not as invoices. Larkfield, which bundles Chromium, depends on the same upstream budget as Microsoft.

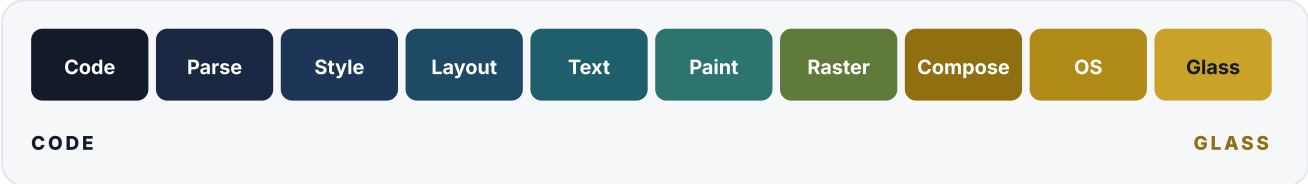
Visibility. Court opinions and audited statements are the only checked numbers in this market; read them the way procurement reads a supplier's accounts. Halbrook's supplier risk file should cite finding 335 and Mozilla's concentration note, not press coverage.

Value. A court remedy or a platform rule upstream can move your roadmap more than any feature release. Corvane's ten-year kiosk plan should name the engine's main payer and what happens to the screen if that payer changes course.

Monday rule: for each engine under a critical screen, write one line: who writes most of it, who pays for that work, and which public record you would watch for a change.

CHAPTER FOURTEEN

The Dark Pages: Antitrust, Exploits and Tracking



The decision this chapter changes is which fonts, images and decoders your critical screens may load at all. Browser code tied into an operating system is where a court found exclusionary conduct, rendering code is where some of the best-documented zero-click attacks started, and where trackers learned to tell one device from another. None of this is in a vendor brochure, but all of it is in court records, CVE entries and published research. This chapter keeps to those records, so you can put the risk in a register instead of a rumor.

The court record: when the browser became part of the OS

The United States and a group of states sued Microsoft in the US District Court for the District of Columbia on 18 May 1998. The court's findings of fact, issued on 5 November 1999, describe how the browser was tied into Windows at the level of the code. Paragraph 161 found that Microsoft "bound Internet Explorer to Windows 95 by placing code specific to Web browsing in the same files as code that provided operating system functions." Paragraph 164 found that "the deletion of any file containing browsing-specific routines would also delete vital operating system routines." Paragraph 174, on Windows 98, found that this mixing "increased the likelihood that a browser crash will cause the entire system to crash."

The court issued conclusions of law on 3 April 2000 and ordered a breakup on 7 June 2000. On 28 June 2001 the D.C. Circuit, sitting en banc, affirmed in part the finding of unlawful monopoly maintenance and held that Microsoft's "commingling of browser and operating system code" was exclusionary conduct in violation of Section 2 of the Sherman Act. It reversed the attempted monopolization finding, vacated the per se tying finding with leave to retry it under the rule of reason, and vacated the breakup remedy. The United States and Microsoft filed a proposed settlement in November 2001 (revised on 6 November). Judge Kollar-Kotelly approved it, with minor changes, by opinion on 1 November 2002, and the

final judgment was entered on 12 November 2002. The case was about commingling browser and operating system code and the conduct around it, not about how the browser drew pages.

Two decades later the question moved from bundling to defaults. In US v. Google (D.D.C. 1:20-cv-03010), the court's memorandum opinion of 5 August 2024 found that "in 2021, Google paid out a total of $26.3 billion in revenue share" under its default search contracts. Chapter Thirteen covers that money. For this chapter the point is narrower: the browser and the defaults around it have twice been the subject of a federal antitrust judgment, and both records are public. Quote them, not commentary about them.

Fonts and media: rendering code as the way in

A font file is not passive data. TrueType fonts contain bytecode instructions that a constrained interpreter runs to fit glyphs to the pixel grid, so a font is active, attacker-controlled input. The Duqu malware, first reported in October 2011, was shown in November 2011 to use that. NVD describes CVE-2011-3402 as a flaw in the "TrueType font parsing engine in win32k.sys in the kernel-mode drivers", exploited "via crafted font data in a Word document or web page". Microsoft published Security Advisory 2639658 on 3 November 2011 and fixed the flaw in bulletin MS11-087 on 13 December 2011. Because the font parser ran in the kernel, a document was enough to reach the most privileged code on the machine.

Microsoft's response was to move the parser. In its words, "with Windows 10, GDI font parsing is no longer performed in kernel mode", but "in a sandboxed user-mode process, fontdrvhost.exe", an AppContainer "granted no Capabilities and minimal privileges". Microsoft presented the change at Black Hat USA 2016 as part of the Windows 10 Anniversary Update mitigations.

Android's turn came in 2015. On 27 July 2015 Zimperium described a set of flaws in `libstagefright`, Android's media framework, reachable through a "specially crafted media file delivered via MMS" and affecting "an estimated 950 million devices". The CVEs included CVE-2015-1538. Google announced monthly security updates for its Nexus devices on 5 August 2015, and published the first Nexus Security Bulletin on 13 August 2015, which noted that the bugs were reachable "most notably [via] MMS and browser playback of media." The monthly Android bulletin that device makers follow today grew out of that incident.

Zero-click: a decoder as the first contact

A **zero-click** attack needs no action from the victim: the device parses a message or attachment on its own, and the parser is the target. Two cases are documented in detail by Citizen Lab, the research group at the University of Toronto, and by Google Project Zero.

FORCEDENTRY. On 13 September 2021 Citizen Lab reported a zero-click iMessage exploit, captured on the phone of a Saudi activist and attributed to NSO Group's Pegasus spyware. Files named ".gif" were actually Adobe PSD files and Adobe PDF files, the PDFs containing a JBIG2-encoded image stream. Apple fixed the flaw, CVE-2021-30860, an integer overflow in CoreGraphics, in iOS 14.8 the same day. On 15 December 2021 Project Zero's Ian Beer and Samuel Groß published their analysis. They noted that "just because the source filename has to end in .gif, that doesn't mean it's really a GIF file", and showed that the exploit combined JBIG2's simple bit operations into a small emulated computer built from more than 70,000 segment commands.

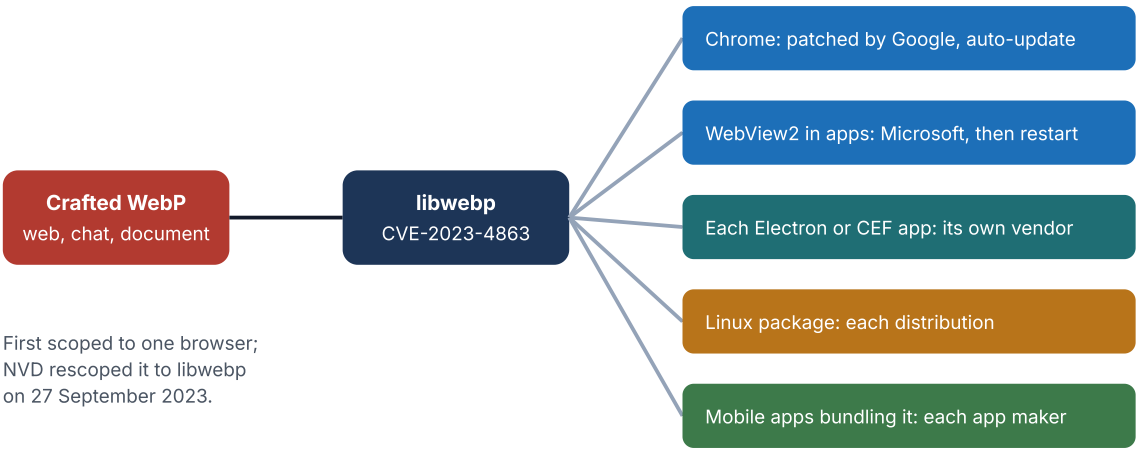
BLASTPASS. On 7 September 2023 Citizen Lab reported another NSO Group zero-click chain, found on the phone of an employee of a Washington, DC based civil society group. It used PassKit attachments containing malicious images sent over iMessage. Apple's iOS 16.6.1, released the same day, fixed CVE-2023-41064, a buffer overflow in ImageIO, and CVE-2023-41061 in Wallet. CISA added both to its Known Exploited Vulnerabilities (KEV) catalog on 11 September 2023.

libwebp. On 11 September 2023 Chrome 116.0.5845.187 fixed CVE-2023-4863, a "heap buffer overflow in WebP", reported by Apple's Security Engineering and Architecture team and Citizen Lab on 6 September; Google said it was "aware that an exploit for CVE-2023-4863 exists in the wild." CISA added it to KEV on 13 September. On 27 September NVD rewrote the description from a Chrome flaw to a flaw "in libwebp in Google Chrome prior to 116.0.5845.187 and libwebp 1.3.2", and a second identifier, CVE-2023-5129, was rejected as a duplicate. The wording change mattered: libwebp is a library that many products bundle, so one flaw had many owners.

Apple's platform answers are on the record too. Project Zero described BlastDoor, "a new, tightly sandboxed" iMessage parsing service in iOS 14, on 28 January 2021. Apple announced Lockdown Mode on 6 July 2022 for iOS 16 and macOS Ventura, describing it as "an extreme, optional protection for the very small number of users who face grave, targeted threats."

FIGURE 14.1

One decoder flaw, many owners



The Frame Ledger's "reachable from untrusted input" column is a question about decoders, not products. One library copy in many products means one flaw with many patchers and many patch dates.

	Windows	macOS and iOS	Linux	Android
Landmark rendering exploit	Duqu, TrueType parser in the kernel (CVE-2011-3402, 2011)	FORCEDENTRY (2021), BLASTPASS (2023), both zero-click on iOS	No single landmark; shared libraries such as libwebp (next row) are patched per distribution	Stagefright media parsing via MMS (2015)
Cross-platform library	libwebp (CVE-2023-4863, 2023): one decoder flaw reached products on every OS that shipped a vulnerable libwebp build and passed untrusted images to its decoder, each patched by its own vendor (Figure 14.1)			
Platform response	Font parsing moved to a user-mode AppContainer, `fontdrvhost.exe`	BlastDoor (iOS 14); Lockdown Mode (iOS 16, macOS Ventura)	Patch per distribution package; bundled copies patched by each vendor	Monthly security bulletins since August 2015
What it means for your screens	Remote fonts on a critical screen are a choice, not a default	High-risk staff can turn on Lockdown Mode	Know which copy each app uses	Bulletin patch level is a device-fleet requirement

Tracking: the renderer as a fingerprint

A **fingerprint** is a set of small differences that, combined, tell one device from another without a cookie. The renderer is a rich source, because the same drawing instructions produce slightly different pixels on different fonts, drivers and GPUs.

- **Canvas.** Mowery and Shacham showed in "Pixel Perfect: Fingerprinting Canvas in HTML5" (W2SP, May 2012) that text and WebGL drawn to a canvas yield device-specific pixels. Acar and colleagues found canvas fingerprinting scripts on "more than 5.5% of the

crawled sites" among the top 100,000 in "The Web Never Forgets" (ACM CCS 2014).

- **Fonts.** Fifield and Egelman, "Fingerprinting Web Users Through Font Metrics" (Financial Cryptography 2015), measured the sizes at which a browser draws glyphs and used them to identify users.
- **GPU.** "DrawnApart" (Laor and colleagues, NDSS 2022) timed GPU execution units and reported "a boost of up to 67% to the median tracking duration", tested on "over 2,500 crowd-sourced devices".

Fingerprinting is invisible to the user and often to the site owner, because it usually arrives inside a third-party script. If your critical screens load third-party scripts, you have accepted their fingerprinting along with them.

Pixel stealing: reading what another site drew

The renderer can also leak what it draws. Paul Stone's "Pixel Perfect Timing Attacks with HTML5" (Black Hat USA, July 2013) showed that a page could time how long the browser took to draw content from another origin and recover it pixel by pixel. Kotcher, Pei, Jumde and Jackson showed related cross-origin pixel stealing with CSS filters at ACM CCS 2013.

A decade later the leak moved into hardware. "GPU.zip" (Wang and colleagues, IEEE Symposium on Security and Privacy 2024, made public in late September 2023) showed that "a malicious webpage can leak pixels from another webpage in the latest version of Google Chrome", because integrated GPUs compress graphical data and the compression changes timing. The attack relied on Chrome rendering cross-origin iframes with cookies and applying SVG filters on the GPU; the researchers reported Firefox and Safari were not affected, and they notified vendors in March 2023. The lesson for architects is that the GPU process and the hardware below it are part of the confidentiality boundary, not only the performance story.

> *Zero-click starts at a decoder. The question is not which products you ship, but which parsers untrusted bytes can reach.*

HOW IT GOT THIS WAY

The richer the renderer became, the larger its attack surface and the better its fingerprint. In the 1990s the court record shows browser code being commingled with the operating system. In 2011 a font parser in the Windows kernel let a document reach kernel privileges. In 2015 a media parser on Android was reachable by text message. In 2021 and 2023 image parsers on iPhones were reached by messages the user never opened, and a widely bundled image library carried an exploited flaw into many products at once. In parallel, research from 2012 to 2022 turned canvas, fonts and GPUs into tracking signals and timing channels. Each platform answered by isolating or rewriting the parser: a user-mode font host on Windows, BlastDoor and Lockdown Mode on iOS, monthly bulletins on Android, and Rust font code in Chrome from 2024 (Chapter Three and Chapter Sixteen). The decision today follows: limit which decoders your critical screens can reach, and know who patches each copy.

What hasn't changed: image decoders. Fonts are moving: Chrome parses web fonts in Rust through Fontations, and Google says FreeType was removed from Blink in Chrome 145 (Chapter Three). Images mostly have not. WebKit's WebP decoder, `WEBPImageDecoder.cpp`, a C++ wrapper around the C library libwebp, was added on 18 October 2010 (WebKit commit `20d7e999518`, then under `WebCore/platform/image-decoders/webp/`) and is still in the tree; Gecko added its own libwebp-based decoder, `image/decoders/nsWebPDecoder.cpp`, on 3 October 2018 (`ea3496614c4a`). Skia's repository has carried an experimental Rust PNG decoder since 9 August 2024 (`cbd5898acb`), moved out of the experimental folder on 17 September 2025 (`3745e62ff9`); this book did not verify which products ship it. Measured 18 November 2026 with `git log --diff-filter=A` on the WebKit, firefox and Skia clones.

THE UNSEEN ANGLE

CVE-2023-4863 was first described as a Chrome flaw. Teams that searched their software inventories for "Chrome" missed the other copies of libwebp they shipped: WebView2 inside their own apps, each Electron and CEF product, distribution packages and mobile apps. NVD's rewrite of the description on 27 September 2023, sixteen days after Chrome's fix, is the documented moment the scope changed. A Frame Ledger row that lists the decoder, not the product, would have found those copies on day one. Source: NVD change history for CVE-2023-4863; Chrome stable release notes, 11 September 2023.

This hop in the Frame Ledger

Lead company: Halbrook Bank, for its teller desktop and its mobile app.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
6 Raster: image decoders in Microsoft's Edge WebView2 runtime (WebP, PNG, JPEG)	Google upstream and library maintainers; Microsoft ships	Microsoft, evergreen; active only after the teller app restarts	Renderer, sandboxed	Teller, capture tools	**RENT** and **ISOLATE**: no outside images on the approval screen; force a restart within a day of an emergency fix
4 Text: fonts on the teller desktop	Bank's font files; Microsoft for `fontdrvhost.exe`	Bank for files; Windows Update monthly	Renderer and user-mode font host	Teller	**OWN** the font files; block remote fonts
6 Raster: mobile app, system image and message parsers (iOS ImageIO)	Apple	Apple, OS update; bank sets minimum OS version	OS services; BlastDoor for iMessage	Customer	**RENT** and **PIN** a minimum OS; offer Lockdown Mode guidance to high-risk staff
0 Source: third-party scripts on the customer web banking pages	Script vendors	Vendor schedule	Page renderer	Script vendor, through fingerprints	**ISOLATE**: no third-party scripts on login, consent and payment pages

SECURITY LENS

Who can change it? Library maintainers write decoder fixes; browser, OS and app vendors package them; your devices run them. A message or web page can reach the decoder without anyone at your company choosing it.

How fast can it be fixed? In the zero-click and libwebp cases above, the platform vendor shipped a fix within days: Apple on the day Citizen Lab published FORCEDENTRY and BLASTPASS, Chrome five days after the libwebp report. Not every case was that fast: Duqu's font flaw took about six weeks from Microsoft's advisory to its bulletin. Your exposure is the slowest bundled copy plus the time until it is running.

What does AI change here? AI-assisted fuzzing finds more parser bugs, which means more advisories against the copies you bundle. AI agents that open attachments and render pages on a user's behalf also add zero-click reach: a decoder no human looked at still parses the file.

BUSINESS IMPLICATIONS

Cost. Each bundled engine is a separate emergency patch. Larkfield Software paid for libwebp in its own release; Halbrook Bank's teller desktop got Microsoft's fix but needed restarts to activate it. Corvane Hotels should know which engine copy each kiosk and tablet runs before the next library flaw, not after.

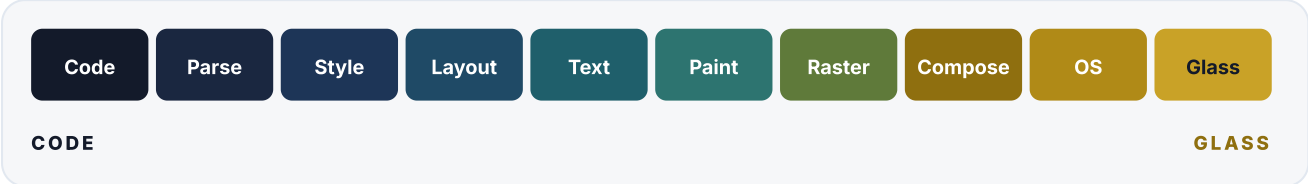
Visibility. Fingerprinting and pixel leaks are invisible to users and often to site owners. Halbrook should inventory third-party scripts on regulated pages and the decoders each screen can reach.

Value. Trust lost in one incident costs more than the engine. A short, documented decoder list for critical screens is evidence a regulator or customer can check.

Monday rule: for each critical screen, list the image and font decoders that untrusted bytes can reach, who patches each copy, and how many days the last emergency fix took to become active.

CHAPTER FIFTEEN

Cost and Debt: The Patch Bill, the Memory Bill, the Legacy Bill



The decision this chapter changes is how you budget a rendering choice, because the price is not on the license. An engine you bundle is free to download and expensive to keep. You pay three bills for it: a patch bill every time upstream fixes a flaw, a memory bill on every device that runs it, and a legacy bill for every old interface you cannot retire. This chapter prices all three for Larkfield Software with stated assumptions, so you can replace them with your own numbers.

The patch bill: one engine, many emergencies

Chromium ships a new major version every four weeks. Google announced the change on 4 March 2021, starting with Chrome 94 in the third quarter of 2021, with an Extended Stable channel every eight weeks for those who need a slower pace. Electron, the framework Larkfield's app is built on, follows Chromium's Extended Stable line: since September 2021 it ships a new major version every eight weeks, and its documentation says "the latest three *stable* major versions are supported by the Electron team". Three majors eight weeks apart give each version roughly six months of support.

That rhythm sets a floor. To stay on a supported engine, Larkfield must move up at least once every 24 weeks or so, counted from the day the major ships, not the day Larkfield adopts it. That is a little over two upgrades a year on average, but two evenly spaced upgrades leave 26-week gaps, so the floor is three a year, and in practice more, because each upgrade brings Chromium changes its app must test. On top of the routine work come the emergencies: when upstream fixes an exploited flaw, every product that bundles its own copy must build, test and ship its own release. Chapter Fourteen showed how one image-decoder flaw in 2023 reached many bundled engines at once. Microsoft's Edge WebView2 runtime, in its evergreen mode, is patched by Microsoft for every app that uses it; an Electron or CEF vendor patches only its own app, and has to do so itself.

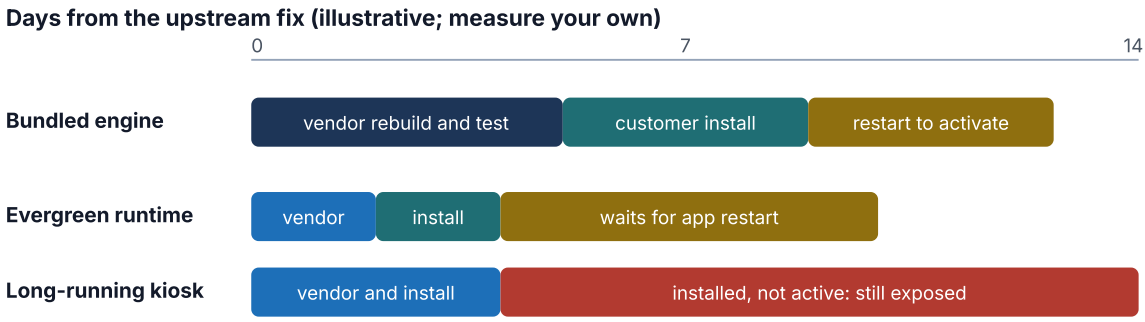
The bill also has a second half that the vendor does not see: **activation**.

ACTIVATION IS NOT INSTALL

A patch counts only when the new code is running. Microsoft's WebView2 documentation says that "if your app runs continuously, your app will continue to use the previous version of the WebView2 Runtime" until it releases "all references to the previous WebView2 environment objects" or restarts. An app that never restarts, like a teller desktop or a kiosk, can have the fix installed for days and still run the old engine. Measure, contract and report the day the fix is running, not the day it was installed.

FIGURE 15.1

The exposure window for one engine flaw



The install SLA ends at the teal bar. The exposure ends only when the new engine is running.

The exposure window has three parts: the vendor's build, the install, and the activation. Contracts and dashboards usually measure the first two.

The memory bill: one engine per app, one process per site

A bundled engine brings its own browser, renderer and GPU processes. Two Electron apps on one laptop share no engine processes; each loads its own copy. WebView2 is different only in part. Its binaries are installed once per machine, and on eligible systems "binaries for Microsoft Edge and the Evergreen WebView2 Runtime are hard-linked together when they are on the same version", which saves disk. But each app, or more precisely each user data folder, gets "a collection of WebView2 Runtime processes", so most of the memory is still per app.

Security adds its own line. Site Isolation, which puts content from different sites in different processes to blunt Spectre-class attacks, was enabled in Chrome 67 on Windows, Mac, Linux and Chrome OS for 99 percent of users. Google's security blog said on 11 July 2018 that

"there is about a 10-13% total memory overhead in real workloads due to the larger number of processes." That is a cost worth paying on screens that load outside content. It is still a cost, and it scales with every engine copy.

No vendor publishes a reliable baseline figure for "memory per Electron app", and the number varies with the content. So Larkfield measures it: private working set of all its processes after an hour of typical use, on a reference laptop per OS, with the method written down. The model below uses an assumed value until that measurement exists.

A worked model for Larkfield

Every input below is an assumption, labeled as such. The arithmetic is the point: replace each number with a measured one and the model still works. The exposure lines use one unit, the **seat-day**: one seat running a flawed engine for one day. A thousand seats exposed for three days is 3,000 seat-days.

Line	Assumption (replace with yours)	Result
Seats	400,000 customer desktops (Larkfield's installed base)	400,000
Memory per running app	Assume 400 MB private memory across all engine processes	160 TB of RAM held across the customer fleet
Share of an 8 GB laptop	400 MB of 8,192 MB	About 5 percent of the machine, for one app
Routine engine upgrades	4 upgrades a year (above the support floor of 3: two evenly spaced upgrades leave 26-week gaps against a support window of about 24 weeks), 3 engineer-weeks each	12 engineer-weeks a year
Emergency engine releases	6 a year, 1 engineer-week each plus release overhead	6 engineer-weeks a year
Patch bill, engineering	Sum of the two lines above	18 engineer-weeks, about 0.4 of a full-time engineer
Exposure per emergency	Simplifying assumption: every seat waits 5 days for the release plus 7 days to install and restart, a fleet mean of 12 days (a median cannot be multiplied by seats). Real fleets have a long tail, so also compute a range of 3 to 30 days per seat	Illustrative: 400,000 × 12 = 4.8 million seat-days (range 1.2 million to 12 million)
Exposure per year	6 emergencies	Illustrative: about 29 million seat-days (range about 7 million to 72 million)

Two lines matter most. The engineering line looks small, which is why the patch bill is easy to underestimate; the real cost is the exposure line, which Larkfield's customers carry. Cutting the ship time from five days to two, or forcing a restart within a day of install, moves the exposure far more than any engineering change. The memory line is paid by the

customer's hardware budget, not by Larkfield, which is why it rarely appears in Larkfield's planning. The 160 TB is not a cash cost by itself: it becomes one when customers buy machines with more memory at their next refresh, or fit fewer sessions on each VDI host.

> *An engine you bundle is free to download and expensive to keep. Your customers pay most of the bill, in memory and in days of exposure.*

The remote-display bill

When a screen is drawn on a server and shown on another machine, as for Halbrook Bank's advisors on virtual desktops (VDI), the frame is rendered twice in effect: once by the engine on the session host, then encoded as video and decoded on the client. Microsoft's Remote Desktop Protocol added "an AVC/H.264 mixed mode" in RDP 8.1 (Windows 8.1 and Windows Server 2012 R2), and a full AVC 444 mode in RDP 10, shipped with Windows 10 version 1511 in November 2015. Administrators enable it with the policies "Prioritize H.264/AVC 444 Graphics mode for Remote Desktop connections" and "Configure H.264/AVC hardware encoding for Remote Desktop connections". Citrix's documentation says "Thinwire is the default display remoting technology", with H.264 or H.265 for the full screen or for changing regions. Its hardware-encoding policy is on by default, and "if such hardware is not available, the VDA falls back to CPU-based encoding using the software video codec."

The cost lesson: without a GPU on the session host, both the engine's drawing and the video encoding land on server CPUs that are priced per session. A web-heavy teller app with animations can cost more host CPU than the business logic behind it. Microsoft also shows how fast a remote graphics option can vanish: RemoteFX vGPU was disabled in the July 2020 security updates and removed from all platforms in the April 2021 updates. Budget the GPU path, and keep a tested CPU path.

The legacy bill: interfaces kept for decades

Compatibility is a promise, and promises cost money every year they are kept. Microsoft's documentation still says "GDI can be used in all Windows-based applications", while advising that for new drawing code "Direct2D should be considered, as it takes better advantage of modern GPUs". Microsoft "is committed to supporting Internet Explorer mode in Microsoft Edge through at least 2029", with at least a year's notice before it ends, although the Internet Explorer 11 desktop app went out of support for certain operating systems on 15 June 2022. On Linux, GNOME 49 disabled its dedicated X11 session by default in September 2025, and GNOME 50 (18 March 2026) shipped Mutter without its

X11 backend (Mutter 50.alpha: "Drop the X11 backend"); apps that depend on X11 "continue to be supported via xwayland". On Android, devices without Google Play depend on the device maker for WebView updates; Chromium's own integration guide warns that the AOSP prebuilt "is not currently updated on a regular schedule, and may have known security issues".

Bill	Windows	macOS	iOS	Linux	Android
Legacy kept alive	GDI and Win32; IE mode in Edge through at least 2029	The legacy WebKit `WebView` class, marked deprecated since macOS 10.14 in favor of `WKWebView`	UIWebView rejected for new apps since April 2020	X11 apps through XWayland after GNOME 49 and 50	Devices without Play rely on the maker for WebView updates
Who carries the patch debt	Microsoft for the evergreen runtime; each Electron or CEF vendor for its own copy	Apple for the OS engine; each bundled-engine vendor for its own copy	Apple, for WebKit in every app that uses it. Bundled engines only with an entitlement: in the EU (iOS 17.4) for browser apps and apps that offer in-app browsing; in Japan (iOS 26.2) for browser apps, and in-app browsing only for browser engine stewards	Each distribution and each bundled-engine vendor	Google for System WebView through Play; the device maker otherwise

HOW IT GOT THIS WAY

Operating system vendors kept old drawing interfaces alive because breaking customers' software costs more than maintaining code. That is why GDI, which dates back to the first versions of Windows, still draws in Windows apps today, and why X11, released by MIT in September 1987, still runs on Linux desktops through XWayland. The second shift came with bundled engines. Electron and CEF let any company ship a browser engine inside its app, which moved the patch cost from the OS vendor to every app maker. Chromium's four-week cadence from 2021, and Electron's eight-week cadence with a three-version window, turned that cost into a fixed calendar. Apple moved the other way on iOS: when it told developers in December 2019 that the App Store would stop accepting new apps using UIWebView from April 2020, the legacy engine's cost came back to the platform owner, although Apple later extended the deadline for app updates beyond the end of 2020.

What hasn't changed: the compatibility promise. As of November 2026 Microsoft's GDI documentation still says it "can be used in all Windows-based applications", and IE mode has a support commitment through at least 2029. The engines keep their own promise too. Quirks mode, the behavior kept for pages written before the standards, still ships a stylesheet in two engines: Gecko's `quirk.css` was added on 2 June 2000 (`a492e0b204da` , "QuirkMode style sheet") and lives today at `layout/style/res/quirk.css` ; WebKit's `quirks.css` arrived on 25 February 2003 (`9d6c8afab70` , under `WebCore/khtml/css/`) and lives today at `Source/WebCore/css/quirks.css` . Measured with `git log --diff-filter=A` and `git ls-tree` on the firefox clone (head `524e4e57a3`) and the WebKit clone (head `f31b755f63d`), 18 November 2026.

THE UNSEEN ANGLE

Most patch SLAs measure install. The number that matters is activation: the day the new engine is actually running. Microsoft documents the gap plainly for WebView2: a continuously running app keeps the old runtime until it releases its environment objects or restarts, and the runtime raises a `NewBrowserVersionAvailable` event so the app can ask the user to restart. A kiosk fleet that reports 100 percent installed can be 0 percent patched. The fix is cheap: handle the event, schedule a restart in a quiet window, and report activation separately from install. Source: Microsoft Learn, WebView2 distribution guide.

This hop in the Frame Ledger

Lead company: Larkfield Software, main app on Windows, macOS and Linux customer desktops.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
1 to 7 Bundled Chromium in Electron	Google upstream; Electron maintainers; Larkfield pays the integration	Larkfield, per release; target 2 days from upstream fix, measured	Browser, renderer and GPU processes per app; renderer sandboxed	Users, capture tools, agents	**OWN** the update channel; stay within Electron's three supported majors
1 to 7 Renderer processes with Site Isolation	Upstream design; customer pays the memory	With the engine	One renderer per site; about 10 to 13 percent memory overhead (2018 figure)	Not applicable	**RENT**: keep it on for screens that load outside content
1 to 7 Activation of engine updates	Larkfield	Restart policy, measured as days to activation	App process	Not applicable	**OWN**: report activation, not install
8 OS compositor on customer VDI	Microsoft or Citrix; customer pays host CPU and GPU	Customer's VDI team, monthly	Session host; codec on GPU or CPU fallback	Remote client, session recording	**RENT**: publish a tested VDI profile with GPU and CPU numbers

SECURITY LENS

Who can change it? Upstream writes the fix, Larkfield packages it, and the customer's machine activates it. Each of the three can delay it, and only the first is outside your control.

How fast can it be fixed? Exposure is the sum of build, install and restart, so a fix counts only when all three are done. Measure all three for the last three engine emergencies and cut the longest first; the restart is often the longest.

What does AI change here? AI-assisted fuzzing and code review are finding more flaws in parsers and engines, which points to more emergency releases a year, not fewer. AI can also shorten the integration and test work for each engine upgrade. The patch bill's engineering line may fall while its count rises.

BUSINESS IMPLICATIONS

Cost. Larkfield should put its patch bill and its customers' memory bill in the product plan, with the model above filled in from measurements. Halbrook Bank, whose advisor VDI hosts have no GPU (Chapter Four), should size them with the advisor app's real CPU rendering and encoding load measured. Corvane Hotels should check that every kiosk engine's support window ends after the kiosk hardware does, or budget the upgrade now.

Visibility. Teams rarely track how many engine copies run on one laptop, or how many days a fix takes to become active. Halbrook and Corvane should report activation separately from install for every evergreen runtime.

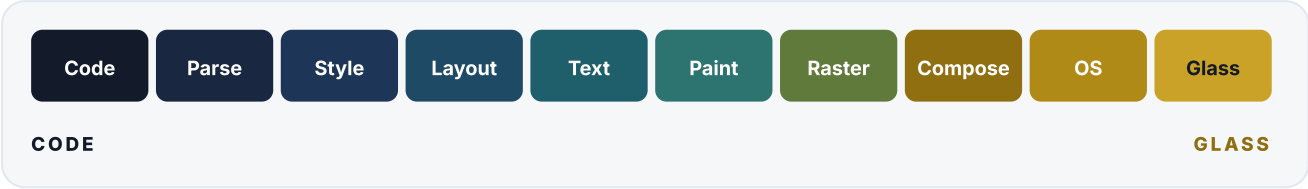
Value. Retiring a legacy interface frees engineers for product work, and a short, measured exposure window is a selling point in security reviews. Larkfield can turn its patch numbers into a promise customers can check.

Monday rule: for your last three engine emergencies, write down the days to build, to install and to activate, and fix the longest one first.

PART V · AHEAD

CHAPTER SIXTEEN

The Screen Ahead: The OS, AI and the Renderer



The decision this chapter changes is how you hedge a screen you will keep for five years. Up to now the book has looked at what is already shipped and on the record. This chapter looks forward, so every forward claim is labeled as opinion and dated, and every fact under it has a source. The short version is that the screen is getting more readers, more generators and more competition for the same chip, while the code that draws it is slowly moving to memory-safe languages. None of that removes the human who has to see and approve the frame that matters.

More readers, none retired

Each wave of computing added a new reader of the pixel and kept the old ones. People read the first desktop screens. Assistive technology became a second reader through accessibility APIs: Microsoft Active Accessibility (MSAA) shipped as an add-on for Windows 95 in 1997, Apple's VoiceOver arrived with Mac OS X 10.4 Tiger in 2005, and UI Automation followed with Windows Vista and .NET Framework 3.0 in 2006. Capture and remote-display tools became a third, and Chapter Nine showed how they copy a composed surface. Software agents that read and drive screens are the newest reader. Chapter Ten covers how they work: pixels, the DOM and the accessibility tree, and what happens when the three disagree. This chapter gives only the outlook.

Our view (as of November 2026): agents will become a normal user of business screens, but regulated decisions will still need a human-visible frame. The practical result is that a critical screen gets two contracts. One is for the human: what was drawn, and what was approved. The other is for the agent: a structured, permissioned representation it may act on. Teams that keep the two in step will ship agent features faster than teams that let agents scrape pixels.

Interfaces that a model writes

A **generated interface** is a screen a model builds at request time instead of one a team designed and tested in advance. The vendors have shipped the plumbing. Anthropic introduced and open-sourced the Model Context Protocol (MCP), a standard for connecting AI assistants to tools and data, on 25 November 2024. OpenAI introduced apps in ChatGPT and a preview Apps SDK, built on MCP, on 6 October 2025. Google Research described "Generative UI", interactive layouts built by Gemini 3 Pro for a single prompt, on 18 November 2025. MCP Apps, a way for a tool to return an interactive interface to the host, was proposed on 21 November 2025 by Anthropic, OpenAI and the MCP-UI project, and was announced as an official MCP extension on the MCP blog on 26 January 2026.

For the rendering pipeline nothing changes: the generated markup still goes through parse, style, layout, text, paint, raster and the compositors. What changes is the source row of the Frame Ledger. Hop 0 used to be code your team reviewed. Now it can be output nobody reviewed, produced seconds before the frame.

Our view (as of November 2026): generated interfaces will spread first in read-only and exploratory screens (reports, dashboards, search answers) and slowly, if at all, in screens that move money or consent. Where they do appear, the generated source should be logged like a transaction record, because a screenshot of a screen that was never designed is weak evidence of what the user was shown.

> *A local model and the compositor share one chip. When both want it in the same 16 milliseconds, one of them waits.*

The model and the frame share one GPU

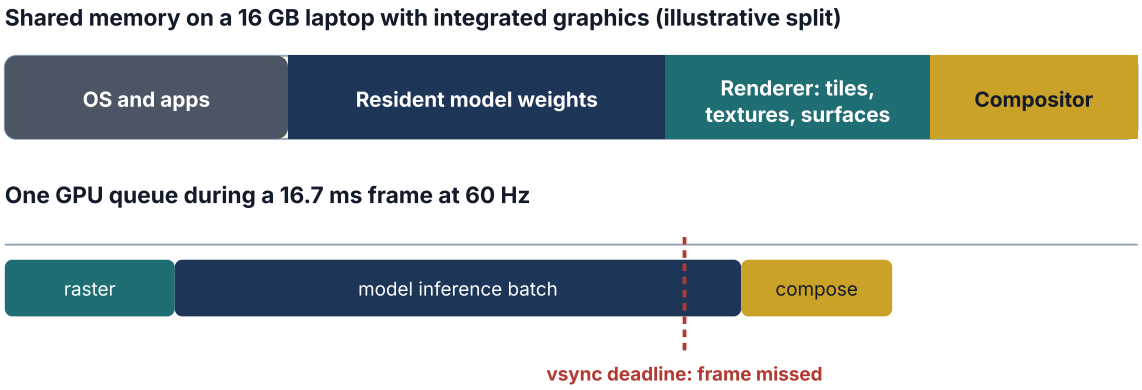
On-device models are now a platform feature, not an experiment. Microsoft introduced Copilot+ PCs on 20 May 2024 with a neural processing unit (NPU) rated at "40+ TOPS" (trillion operations per second), and its hardware requirements page lists 16 GB of memory as the minimum. Apple announced the Foundation Models framework at WWDC25 on 9 June 2025, giving apps access to its on-device model; Apple's technical report describes it as about 3 billion parameters. On Android, Gemini Nano runs inside the AICore system service. Chrome's built-in Prompt API, in its documentation as last updated on 21 October 2026, asks for at least 22 GB of free disk and either a GPU with "strictly more than 4 GB of VRAM" or 16 GB of RAM and four CPU cores.

Those numbers matter to rendering because of **memory residency**: a model's weights must stay loaded in memory to answer quickly, and on many devices that memory is the same pool the renderer and compositor use. Apple's Metal documentation says its GPUs use

"a unified memory model in which the CPU and the GPU share system memory". On a laptop with integrated graphics, the same is broadly true. On these unified-memory devices, memory held by a model is memory that tile caches, textures and window surfaces cannot use. A desktop with a discrete GPU has separate video memory, so the squeeze there is mainly on that card's VRAM when the model runs on it. When the model also runs on the GPU rather than an NPU, the frame's raster and compose work queue behind inference work on the same hardware.

FIGURE 16.1

One memory pool, two tenants



Sizes are illustrative, not measured. An NPU moves inference off the GPU queue but not out of the memory pool.

When a model and the renderer share memory and a GPU, a model that holds memory squeezes the renderer's caches, and an inference batch on the GPU can push compose work past the deadline.

Our view (as of November 2026): GPU and memory contention between local models and the frame will become a device-sizing question for end-user computing teams within two hardware refresh cycles. Expect operating systems to add clearer priority between interactive graphics and background inference, and plan to measure dropped frames with a model loaded, not on a clean machine.

The browser engine as the app layer

Part III showed how many desktop apps already run inside an embedded browser engine. The engine keeps taking on work that used to belong to the operating system. WebGPU, a web API for modern GPU graphics and compute, shipped by default in Chrome 113 in 2023 on ChromeOS, macOS and Windows; in Firefox 141 on Windows in July 2025; and in Safari

26 in 2025 on macOS, iOS, iPadOS and visionOS, after more than a year in Safari Technology Preview. With it, a web page can run both the rendering and compute shaders on the GPU, including model inference, inside the engine's sandbox.

Our view (as of November 2026): for most business software, the browser engine is already the real app layer, and the native operating system is the host that schedules, composes and secures it. That makes the engine's update channel, Chapter Eleven's main question, the most important line in many Frame Ledgers. It also means the GPU process, which Chapter Seven described as a weaker sandbox because it must talk to the driver, carries more of your product than it did five years ago.

Memory-safe renderers, one component at a time

The memory-safety shift in rendering is happening component by component, not engine by engine. The record so far:

- Firefox shipped Stylo, its Rust style engine, by default in Firefox 57 (November 2017), and began rolling out WebRender, its Rust GPU renderer, in Firefox 67 (2019) to a subset of Windows 10 users with NVIDIA graphics.
- Google announced support for Rust in Chromium on 12 January 2023. Its Fontations project replaced the parts of FreeType that Skia uses with a Rust library, Skrifa: first in Chrome 128 (August 2024) for less common formats, then in Chrome 133 (February 2025) for all web fonts on Linux, Android and ChromeOS, and as a fallback for web fonts on Windows and Mac; Google says FreeType was later removed from Blink in Chrome 145 (Chapter Three).
- Google reported that memory-safety bugs fell from 76 percent of Android vulnerabilities in 2019 to 24 percent in 2024 (September 2024), and below 20 percent for the first time in 2025 (November 2025).
- Servo, the Rust browser engine, joined Linux Foundation Europe on 7 September 2023, began binary releases in October 2025 (Chapter Twelve), and released the `servo` crate as an embeddable library (v0.1.0) on 13 April 2026.
- Vello, a Rust 2D renderer from the Linebender project, is still pre-1.0. Its README calls the GPU compute renderer "experimental" and says its CPU renderer is "currently overall more mature".

These are components shipping in specific products, not drop-in replacements for Blink or WebKit. They also move the hardest parsing first. Fonts and image decoders are exactly where Chapter Fourteen's exploits started.

Our view (as of November 2026): within five years the first contact with untrusted fonts and images in the major engines will be mostly memory-safe code, while layout, the DOM and GPU drivers stay largely C and C++. Ask your engine vendors which decoders are memory-safe, and prefer those for screens that load outside content.

Provenance for pixels

As models generate more images, video and interfaces, the question "where did this pixel come from" gets a technical answer. C2PA (the Coalition for Content Provenance and Authenticity) publishes a specification for signed metadata, called Content Credentials, that travels with a media file; the published PDF of version 2.3 is dated 5 January 2026 (its version history lists it as December 2025). Google announced at the Pixel 10 launch on 20 August 2025 that its "latest phones will be the first to have industry-standard C2PA Content Credentials built into the native camera app", and its security blog said on 10 September 2025 that Pixel 10 is "the first to have Content Credentials built in across every photo created by Pixel Camera". In the EU, Article 50 of the AI Act requires providers of systems that generate synthetic content to mark it in a machine-readable form from 2 August 2026. Regulation (EU) 2026/1744 of 8 July 2026, the Digital Omnibus on AI, in force from 27 July 2026, gives providers of systems already on the market before 2 August 2026 until 2 December 2026 to comply with that marking duty, Article 50(2).

Provenance describes a file. It does not prove what a panel displayed or what a person saw. That gap is the trusted-pixel problem from Chapter Nine, and it stays open.

Stated direction (dated)	Windows	macOS	iOS	Linux	Android
On-device model runtime	Copilot+ PCs, NPU 40+ TOPS, 16 GB minimum (May 2024)	Foundation Models framework, about 3B on-device model (June 2025)	Foundation Models framework, same on-device model (June 2025)	No single vendor runtime; distribution and app choice	Gemini Nano in AICore system service
Memory-safe rendering parts	Chrome: Rust fonts (Fontations) as the web font fallback where the system cannot read a format (Chrome 133, 2025); FreeType removed from Blink (Chrome 145)	Same in Chrome; Safari's own policy not published in comparable form	Browsers use WebKit outside the EU and Japan entitlements, so Chrome's Rust font path does not apply; Apple publishes no comparable policy	Fontations for all web fonts in Chrome 133 and system fonts in Chrome 136; FreeType removed from Blink in Chrome 145; Servo and Vello available as components	Fontations for web fonts in Chrome 133 and system fonts in Chrome 136; FreeType removed from Blink in Chrome 145; memory-safety bugs below 20 percent (2025)
Pixel provenance	EU AI Act marking duty from 2 Aug 2026 applies to providers, not to the OS	Same EU duty	Same EU duty	Same EU duty	Pixel 10 camera adds C2PA Content Credentials to its photos (announced 20 Aug 2025)

HOW IT GOT THIS WAY

The screen gained readers in layers. MSAA (1997) and then UI Automation (2006) gave assistive technology a structured view of Windows, and VoiceOver (2005) built a screen reader into the Mac. Chrome split rendering into sandboxed processes in September 2008, and Site Isolation followed in Chrome 67 in 2018, both because the renderer had become the most exposed parser on the machine. In 2023 Google opened Chromium to Rust, Servo moved to Linux Foundation Europe, and WebGPU shipped in Chrome. In 2024 and 2025 the operating systems added on-device models as a platform service. Each step added a new owner or reader to the frame and retired none. The decision today follows from that: plan for more readers of every critical screen, not fewer.

What hasn't changed: a human still has to see and approve the decisive frame, and the code that draws it still mostly predates memory-safe languages. Servo shows what the alternative costs: in its repository (`github.com/servo/servo`, clone at commit `ff0b018`, 18 November 2026) the `components/` tree holds 1,605 Rust files and no C or C++ files. Its commit history, measured in Figure 12.1, shows fourteen years of work to reach an embeddable pre-1.0 library.

THE UNSEEN ANGLE

Most AI hardware discussion is about model speed. The cost that reaches the user is the frame that model steals. On unified-memory devices, Apple's own Metal documentation says the CPU and GPU share system memory, so a resident model shrinks what the renderer can cache, and inference on the GPU sits in the same queue as raster and compose work. No vendor publishes a frame-drop budget for "model loaded" versus "model idle". A team that measures it on its reference devices will know something its competitors are guessing. Source: Apple, "Choosing a resource storage mode for Apple GPUs"; Chrome Prompt API hardware requirements.

This hop in the Frame Ledger

Lead company: Halbrook Bank, which is adding an advisor assistant to the teller and advisor desktop and keeping the payment-approval screen.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
0 Source: assistant-generated summary panel	Bank's AI platform team; model vendor	Bank prompt and template release; model updates on vendor schedule	Renderer of Microsoft's Edge WebView2 runtime, sandboxed	Advisor, logging, screen agents	**OWN** the template and log every generated panel; never on the approval screen
4 Text: web fonts on the assistant panel	Google upstream, Microsoft ships	WebView2 evergreen; activation on app restart	Renderer, sandboxed	Advisor	**RENT** the engine; block remote fonts, use bank font files
6 to 7 Raster and engine compositor beside a local model	Google upstream; GPU maker; OS vendor for the model runtime	Evergreen runtime; driver pinned by the bank; OS monthly	GPU process; model runtime in its own service	Not applicable	**PIN** drivers; measure dropped frames with the model loaded
10 Glass and readers: approval screen	Bank owns the policy	Policy change, days	Not applicable	Teller, customer, any camera, agents if allowed	**ISOLATE**: no agent actions on the approval frame; human approves amount and payee as text

SECURITY LENS

Who can change it? Model vendors and your own prompt authors now write part of the screen's source. The engine vendor still owns how it is drawn, and the OS vendor owns the model runtime and the compositor.

How fast can it be fixed? A bad generated panel can be fixed as fast as your template and logging let you find it. A flaw in a decoder or the GPU process is fixed at the speed of the engine channel, and memory-safe components lower how often that emergency happens. They do not remove it.

What does AI change here? Agent permissions become screen permissions: whatever an agent can read or click on a screen, it can misuse. Generated source adds a new untrusted input at hop 0. Our view (as of November 2026): treat generated interface code like third-party content, rendered in the same sandbox and allowed the same decoders, never more.

BUSINESS IMPLICATIONS

Cost. Larkfield Software should size its minimum desktop spec with a local model loaded, because its customers' laptops will run one beside Larkfield's engine. Corvane Hotels should not add on-device models to kiosks without testing heat, memory and frame drops on the exact kiosk hardware.

Visibility. Halbrook Bank should log every generated panel and every agent action as records tied to the frame they affected. Without that, an auditor sees a screenshot and nothing else.

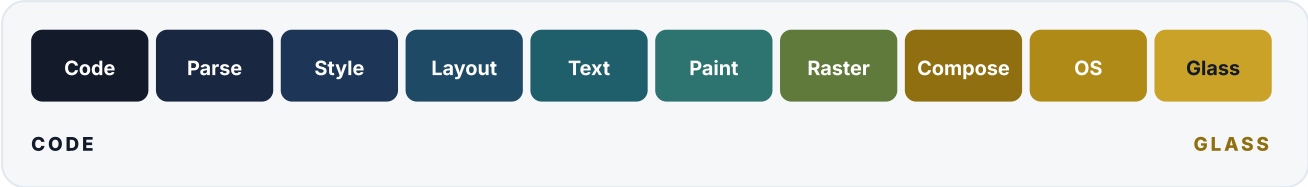
Value. Screens that agents can read safely and humans can verify become a product feature. Larkfield can sell a structured, permissioned view of its app to customers' agents; Halbrook can show a regulator that a human approved the decisive frame.

Monday rule: for each critical screen, write down which parts a model may generate, which an agent may act on, and which only a human may approve, and measure one frame with a local model loaded.

CLOSING

EPILOGUE

From Reading the Frame to Owning It



This book followed one frame from code to glass and stopped at every hop to ask three questions: who wrote this, who pays for it, and who can fix it when it breaks. The answers were rarely the team that shipped the screen. That is not a failure. It is how modern screens are made, and it is a decision a company can now take on purpose instead of by accident.

What you now see on every screen

A screen used to look like one thing: your app. It now looks like a chain. Markup or code is parsed, styled and laid out. Text is shaped and its glyphs are drawn. A display list is painted, rastered into tiles, and handed to an engine compositor and then, in most cases, to the operating system's compositor before the display scans it out to the panel. Native apps, hardware overlays and direct flip can skip part of that chain. Web and cross-platform stacks usually walk all of it.

Each link has an owner. On Windows the frame of a Halbrook teller screen passes through code written largely by Google, shipped and patched by Microsoft, drawn through a GPU driver from a third vendor and composed by the Desktop Window Manager. On a Mac or an iPhone, Apple owns most of the chain. On Linux, the chain is a set of open projects, each with its own maintainers. On Android, the chain depends on Google, the device maker and how long the device keeps getting updates.

And each link has readers. The person in front of the glass is only one of them. Capture APIs, remote display clients, assistive technology and now AI agents read the same frame, each through its own path, and none of them is guaranteed to see exactly what the panel lit. A screenshot is a copy of a composed surface, not a record of the glass.

Seeing this changes the questions a CTO asks. "Electron or native?" becomes "who patches this hop, through which channel, and how many days does it take to reach our devices?" "Is it secure?" becomes "which hops can untrusted input reach, and which process runs them?"

"Can we prove what the customer saw?" becomes "what did we render, what did the accessibility tree say, and what did the customer sign?"

> *You do not have to own the frame. You have to know who does, and how fast they can fix it.*

Five decisions to take this quarter

None of these needs a new platform or a rewrite. Each needs a named person and a date.

Decision	What it takes	Who decides
1. Fill one Frame Ledger for your most critical screen	Pick the one screen where a wrong pixel costs the most: a payment approval, a consent, a kiosk check-in. Trace one frame on one reference device per OS and fill the core worksheet in Appendix B. Name an accountable person for every row.	Chief architect, with the product owner of that screen
2. Put the patch channel into the next runtime contract	For every engine you ship or embed, write down the update channel and the measured days from an upstream fix to a fix on your devices. For an evergreen runtime, add an activation target, not only an install target: an update is not live until the app restarts or releases the old environment.	CTO, with procurement
3. Decide who may read regulated screens	List every capture path, recording tool, remote display client and agent that can read the critical screen. Decide which are allowed. Treat display affinity and secure-window flags as deterrents with stated limits, not as controls.	CISO, with compliance
4. Put an end date on every pin	List each pinned engine and runtime version in the fleet, with the vendor's support end date. A pin without an end date is debt with no repayment plan.	Head of end-user computing

5. Measure three numbers before the next budget	The patch delay for the last three critical CVEs on your engine hop, the engine memory per seat on a reference device, and the frame time of the critical screen at the refresh rate your devices really run.	VP Engineering, with FinOps

The Ledger turns these five into one page. Its last column asks for one of four words. **Own** a hop when you can fix it faster than anyone else and can afford to keep doing so. **Rent** it when the owner patches well and you can see their channel. **Pin** it when stability matters more than freshness, and only with an end date. **Isolate** it when you cannot trust it and cannot replace it. In practice that means concrete mechanisms: a process sandbox that stays switched on (no sandbox-disabling flags), a separate view, window or desktop for the critical screen, and no remote fonts or third-party script on it. Owning more hops is not automatically safer. Every hop you own is a hop you must patch.

Where the three companies landed

Larkfield Software stopped treating its bundled engine as a free part of the build. It now tracks, for each release of its desktop app, how many days it trails the engine's upstream security fixes, and it reports that number to its board next to uptime. It kept the bundled engine for the main app, because the consistency is worth the patch bill to its customers, and it is piloting its hottest view, the analytics canvas, as native code drawn with Vello on wgpu on Windows and macOS, where frame time and memory per seat matter most; it will move the view only when a trace on a reference desktop shows the gain (Chapter Twelve).

Halbrook Bank filled its first Frame Ledger for the branch payment-approval screen. It kept Microsoft's Edge WebView2 runtime in evergreen mode, set a restart window so updates become active, blocked remote fonts on that screen, and removed screen agents and recording tools from the approval desktop. It stopped signing off on screenshots: the amount and payee are rendered in plain text and checked against the transaction record.

Corvane Hotels listed every pinned engine on its kiosks and tablets with its support end date, and found two that had already passed. It put an end date on every pin, added a display panel list to its kiosk purchasing standard, and now buys tablets with a stated update period. Its signage on Linux is the one fleet where it chose to own more, because it already had the people to patch it.

None of them owns the whole frame. Each now knows who does.

The pixel your user trusts

Book 1 of this series ended at the human eye, the one layer no vendor ships. This book ends one step before it, at the last pixel that a system can still control. Between your code and that pixel stand many owners. Most of them do good work, and most of them will never know your name. The job is not to replace them. The job is to know them, to measure them, and to decide, hop by hop, which ones you trust with the screen your customer believes.

> **ONE STEP FURTHER: THE PIXEL RUNTIME**
>
> I went one step further than this book. I built my own runtime, the Pixel Runtime, and it now draws my own IDEs, apps and command-line tools under the jc10 name. Writing about the frame and drawing it yourself teach the same lesson from two sides: every hop you own is yours to fix.
>
> A later book in this series goes deeper into how it is built. This book stays on the rendering layer. You can follow the work at https://jc10.ai/.

APPENDICES

APPENDIX A

Glossary

These are the terms this book uses, in plain words, in alphabetical order. Each entry names the chapter where the term matters most; the chapter gives the full story, the dates and the sources. Where a definition depends on a version or a date, it is as of November 2026.

A to D

Term	Meaning	Chapter
Accessibility tree	The structured description of a screen (roles, names, values, states) that the system exposes to assistive technology. Screen readers read it, and AI agents now read it too. It can disagree with the pixels.	Chapter Ten
Activation (of an update)	The moment an installed update actually runs. An evergreen runtime can be installed on a device and still not be active until the app restarts or releases the old environment.	Chapter Eleven
Android Protected Confirmation	An Android feature (Android 9 and later, on supported devices) that shows a short prompt in a protected dialog and signs the prompt text the user confirmed with a hardware-backed key. It proves the user confirmed that text; it does not prove what else was on the screen.	Chapter Nine

Android System WebView	The component that lets Android apps show web content. It is built from Chromium and updated through the app store on devices with Google Play.	Chapter Eleven
ANGLE	A Chromium library that translates OpenGL ES calls into the platform's native graphics API, such as Direct3D on Windows.	Chapter Four
Blink	The rendering engine of the Chromium project, forked from WebKit by Google in April 2013. It runs in Chrome, Edge, WebView2, Electron and the Android System WebView.	Chapter Two
Bundled engine	A browser engine shipped inside each app, as Electron and CEF apps do. The app vendor, not the OS, decides when it is updated.	Chapter Eleven
Capture path	Any route by which a copy of the frame leaves the glass: screenshot APIs, screen recording, remote display, screen sharing. The Frame Ledger has a row for it.	Chapter Nine
CEF	Chromium Embedded Framework: an open-source framework for embedding Chromium in desktop apps.	Chapter Eleven
Compositor	The code that takes separately drawn layers or surfaces and combines them into one image. An engine compositor combines the layers of one page or app; an OS compositor combines the windows of the whole screen.	Chapter Five

Core Animation	Apple's layer and animation framework on macOS and iOS. Apps build a tree of layers, and the system renders and composites them.	Chapter Five
Core Text	Apple's text layout and font technology on macOS and iOS.	Chapter Three
CVE	Common Vulnerabilities and Exposures: the public identifier given to a known security flaw, for example CVE-2023-4863 in libwebp.	Chapter Fourteen
DirectWrite	The Windows text API. Inside Chromium on Windows it supplies font enumeration, metrics and glyph rasterization; shaping is done by HarfBuzz.	Chapter Three
Display affinity	A Windows window setting (`SetWindowDisplayAffinity`) that asks the system to keep a window out of supported capture paths. Microsoft states it is not a security feature or DRM.	Chapter Nine
Display list	A recorded list of drawing commands (draw this rectangle, this glyph run, this image) produced by paint and replayed by raster.	Chapter Four
DOM	Document Object Model: the tree of elements the parser builds from HTML, which scripts read and change.	Chapter Two

DWM	Desktop Window Manager (`dwm.exe`), the Windows compositor. Desktop composition arrived with Windows Vista; since Windows 8 it is always on.	Chapter Five
Dynamic linking (PSD2)	The EU rule (Delegated Regulation (EU) 2018/389, Article 5) that the payer must be made aware of the amount and payee of a payment being authenticated. It is not a display attestation.	Chapter Nine

E to L

Term	Meaning	Chapter
Electron	An open-source framework that builds desktop apps from web code by bundling Chromium and Node.js in each app.	Chapter Eleven
Engine family	A lineage of browser engines. Since January 2020 there are three in wide use: Blink, WebKit and Gecko.	Chapter Thirteen
Evergreen runtime	A runtime the vendor keeps updated on the device without the app vendor shipping a new build, such as Microsoft's Edge WebView2 runtime in its evergreen mode.	Chapter Eleven
FLAG_SECURE	An Android window flag that keeps a window out of supported screenshots and off non-secure displays. It does not stop a camera, an accessibility service reading the view tree, or a compromised device.	Chapter Nine

Flip presentation	A way of handing a finished frame to the display by swapping buffers instead of copying them. On Windows this is the DXGI flip model; with independent flip the compositor can step aside for a full-screen surface.	Chapter Five
Frame Ledger	This book's worksheet: one row per hop of one frame of one critical screen, recording owner, payer, patch channel, process, readers and a decision. The blank is in Appendix B.	Chapter One
FreeType	An open-source library that reads font files and rasterizes glyphs, used on Linux, Android and inside many apps.	Chapter Three
Gecko	Mozilla's browser engine, used by Firefox.	Chapter Two
Glyph	The drawn shape of a character or group of characters in a given font.	Chapter Three
GPU process	The separate process in Chromium (and in WebKit) that talks to the graphics driver for rasterizing and compositing. It is sandboxed, but more weakly than renderers, because it must reach the driver.	Chapter Seven
HarfBuzz	The open-source text shaping engine. Chromium uses it on every platform, including Windows; Android and Linux desktops use it too.	Chapter Three

Hardware Composer (HWC)	Android's hardware abstraction for the display. SurfaceFlinger asks it how each layer should be composed, often on display planes instead of the GPU.	Chapter Five
Hit-testing	Deciding which element or window is under a point that was touched or clicked.	Chapter Eight
Hop	One stage a frame passes through on its way to the glass. The book numbers them 0 to 10: 0 Source, 1 Parse, 2 Style, 3 Layout, 4 Text, 5 Paint, 6 Raster, 7 Engine compositor, 8 OS compositor, 9 Scanout and display, 10 Glass and readers, plus two side rows, Input and Capture. Sub-rows are written 4a, 4b and so on.	Chapter One
Input-to-photon latency	The time from a touch, click or key press to the change being lit on the panel. It usually spans two to three refresh intervals.	Chapter One
Isolate	A Frame Ledger decision: keep a hop you cannot trust or replace inside a boundary (a sandbox, a separate process, a separate desktop) so its failure cannot spread.	Chapter One
KHTML	The KDE project's HTML engine. Apple's WebKit forked from it, and Blink forked from WebKit, so both descend from it; little of its original code survives.	Chapter Two
KMS	Kernel mode setting: the Linux kernel interface (part of DRM) that sets display modes and puts buffers on screen planes.	Chapter Six

Layout	The hop that turns styled elements into boxes with positions and sizes.	Chapter Two
LayoutNG	Blink's current layout engine, which replaced its legacy layout engine.	Chapter Two

M to R

Term	Meaning	Chapter
MediaProjection	The Android API (API level 21 and later) through which an app, with the user's consent, captures the screen.	Chapter Nine
Minikin	Android's text layout library, which uses HarfBuzz for shaping and ICU for Unicode rules.	Chapter Three
Multiplane overlay (MPO)	Display hardware that scans out several surfaces on separate planes, so the compositor does not have to draw them into one image first.	Chapter Five
Own	A Frame Ledger decision: your team writes or forks the hop and patches it. Only worth it when you can fix it faster than the current owner and keep doing so.	Chapter One
Paint	The hop that walks the laid-out boxes and records what to draw as a display list.	Chapter Four
Parse	The hop that reads markup or code and builds a tree, such as the DOM.	Chapter Two

Patch channel	The route a fix takes to your devices (OS update, app store, evergreen runtime, your own release) and the time it takes.	Chapter Fifteen
Pin	A Frame Ledger decision: hold a hop at a known version for stability. It must come with a support end date.	Chapter One
Pixel Bill of Materials (PBOM)	Book 1's inventory of what is on the box and who owns it. Each Frame Ledger row points to the PBOM row of its component.	Chapter One
PipeWire	The Linux media server that carries screen capture streams on Wayland desktops, through the desktop portal.	Chapter Nine
Raster	The hop that turns a display list into pixels, usually in tiles and usually on the GPU.	Chapter Four
Refresh interval (vsync)	The time between two display refreshes: about 16.7 ms at a fixed 60 Hz, about 8.3 ms at 120 Hz. Variable refresh changes it frame by frame.	Chapter One
Renderer process	The sandboxed process in a multi-process browser that parses, styles, lays out and paints web content.	Chapter Seven
Rent	A Frame Ledger decision: use the hop as its owner ships it, while watching the owner's patch channel.	Chapter One

S to Z

Term	Meaning	Chapter

Sandbox	A set of operating system restrictions that limits what a process can open, call or reach, so a flaw in it does less harm.	Chapter Seven
ScreenCaptureKit	Apple's screen capture framework, introduced in macOS 12.3. Capture requires the user's screen recording permission.	Chapter Nine
Servo	A browser engine written in Rust, started at Mozilla and now hosted by Linux Foundation Europe.	Chapter Twelve
Shaping	Choosing which glyphs to draw for a run of text, and where, following the font's rules and the script's needs (ligatures, joining, reordering).	Chapter Three
Site Isolation	Chromium's policy of putting different sites in different renderer processes, to limit what a compromised renderer can read.	Chapter Seven
Skia	The open-source 2D graphics library behind Chrome, Android and Flutter, among others.	Chapter Four
Style	The hop that applies CSS rules to every element and computes its final style values.	Chapter Two
SurfaceFlinger	Android's system compositor. It wakes on each refresh, collects the visible surfaces and composes them, with help from the Hardware Composer.	Chapter Five

Tauri	An open-source framework for desktop and mobile apps that uses the operating system's own web view instead of bundling an engine.	Chapter Eleven
TrueType bytecode	Instructions inside a TrueType font that adjust glyph outlines, run by a constrained interpreter in the font engine. It makes a font active, attacker-controlled input, not executable code.	Chapter Three
UI Automation	The Windows accessibility API, shipped in 2006 with Windows Vista and .NET Framework 3.0. Assistive technology and automation tools read the screen through it. It followed Microsoft Active Accessibility (MSAA), first released for Windows 95 in 1997.	Chapter Ten
Viz	Chromium's display compositor service, which aggregates compositor frames from all processes and draws them with the GPU.	Chapter Five
Wayland	The Linux display protocol in which the compositor is also the display server. Mutter (GNOME), KWin (KDE) and wlroots-based compositors implement it.	Chapter Five

WebKit	Apple's open-source browser engine, behind Safari, WKWebView and WebKitGTK. On iOS, apps must use it, except apps holding Apple's alternative browser engine entitlements: in the EU (from iOS 17.4), browser apps and apps that offer in-app browsing; in Japan (from iOS 26.2), browser apps, and in-app browsing only for apps from browser engine stewards.	Chapter Eleven
WebView2	Microsoft's Edge WebView2 runtime for embedding web content in Windows apps. Microsoft ships and patches it, on an upstream largely written by Google.	Chapter Eleven
WindowServer	The macOS process that composites the windows of all apps onto the display.	Chapter Five

APPENDIX B

The Frame Ledger, Blank

This is the worksheet the book fills chapter by chapter. Copy it, one per critical screen. The core worksheet fits on one page and is the one you bring to a decision meeting. The extended sheet holds the evidence behind it: measured times, memory, tools and tested recovery. Keep both, and keep them next to the Pixel Bill of Materials from Book 1, because every row here points to a row there.

Header: one per screen

Field	Fill in
Screen	The one critical screen this Ledger covers (for example: branch payment approval, kiosk check-in, consent dialog)
Why it is critical	What a wrong, late or copied pixel on this screen would cost: money, consent, safety, regulation
Stack	Evergreen shared runtime, bundled engine, OS engine or native stack, with the version policy
Reference devices	One per OS, with OS version, GPU and display refresh rate
Accountable owner	One named role for the whole Ledger
Date filled, next review	The date, and a review date no more than six months later

The core worksheet (6 columns)

Rows 0 to 10 follow the frame from code to glass. The two side rows follow the input back from the glass and every copy of the frame out of it. Leave a cell blank rather than guess; a blank is a finding.

Hop and component	Owner and who pays	Who patches, and how fast	Process and sandbox	Who can read it	Decision (Own / Rent / Pin / Isolate)
0 Source (markup, code or native scene)					
1 Parse					
2 Style					
3 Layout					
4 Text (shaping, fonts, glyph raster)					
5 Paint (display list)					

6 Raster						
7 Engine compositor						
8 OS compositor						
9 Scanout and display (timing, color, HDR)						
10 Glass and readers (who sees the pixels)						
Input (glass to hit-test to handler)						
Capture (every path that copies the frame out)						

The extended sheet (12 columns)

Twelve columns do not fit across an A4 page, so the sheet is printed in two halves. Use the same 13 rows as the core worksheet in both halves (add sub-rows such as 4a and 4b where one hop has two components); in a spreadsheet, keep them as one sheet.

Hop	1 Component and version policy	2 PBOM row	3 Owner (writes it)	4 Who pays	5 Who patches, channel, typical delay	6 Process and sandbox
0 Source						
1 Parse						
2 Style						
3 Layout						
4 Text						
5 Paint						
6 Raster						
7 Engine compositor						
8 OS compositor						
9 Scanout and display						
10 Glass and readers						
Input						
Capture						

Hop	7 Reachable from untrusted input?	8 Frame cost	9 Observable by	10 Who can read the pixels	11 Tested recovery	12 Decision
0 Source						
1 Parse						
2 Style						
3 Layout						
4 Text						
5 Paint						
6 Raster						
7 Engine compositor						
8 OS compositor						
9 Scanout and display						
10 Glass and readers						
Input						
Capture						

What goes in each column

Column	What goes in it
1 Component and version policy	The real component (for example Blink LayoutNG, DirectWrite, DWM, SurfaceFlinger) and whether it is evergreen, pinned, tied to the OS or bundled in the app.
2 PBOM row	The row of the Book 1 Pixel Bill of Materials for that component. Link it; do not copy it.
3 Owner (writes it)	The upstream project or vendor that writes the code.
4 Who pays	Who funds the work: a vendor, a foundation, a device maker, or you. Write "not published" when no record says.
5 Who patches, channel, typical delay	The update channel (OS update, app store, evergreen runtime, your own release) and the measured days from an upstream fix to the fix being active on your devices.
6 Process and sandbox	Which process runs the hop (renderer, GPU process, app, OS compositor) and how strongly it is sandboxed.
7 Reachable from untrusted input?	Yes or no, and from what: remote fonts, images, HTML, script, IPC from a less trusted process.
8 Frame cost	Measured time for this hop (median and 95th percentile over repeated runs, at the device's real refresh rate), memory in MB, and energy where you can measure it. A single frame is a sample, not a measurement.

9 Observable by	The tool that sees this hop: Perfetto or Chrome tracing, Windows Performance Analyzer, Xcode Instruments, Android `dumpsys gfxinfo`, or "nothing".
10 Who can read the pixels	The user, OS capture APIs, recording and remote display tools, assistive technology, AI agents, protected media paths.
11 Tested recovery	The tested time to roll back or fail over this hop, with the date of the test, or "untested".
12 Decision	A capitalized tag first, OWN, RENT, PIN or ISOLATE, then a one-line reason and the name of the accountable person. When two tags apply, put the primary one first (for example, RENT and ISOLATE).

How to fill it

- **Pick one screen and one reference device per OS.** Start with the screen where a wrong pixel costs the most. One Ledger, done well, teaches more than ten done in a hurry.
- **Trace a frame with the platform tool.** Run the screen's real workload several times and record, for each hop, the process, the time and the memory. Write percentiles, not one number.
- **Fill owner, payer and channel** from vendor documents, release notes and the PBOM.
- **Mark untrusted inputs and the readers of the pixels:** every capture path, recording tool, remote display client, assistive technology and agent that can see the screen.
- **Measure the patch delay** for the last three critical CVEs on each hop you can: the date the upstream fix shipped, and the date it was active on your devices.
- **Decide, per row, and name a person.** A row with a decision and no name is not decided.

> **THE FOUR DECISIONS**
>
> **Own:** you write or fork the hop and patch it yourself. Choose it only when you can fix it faster than the current owner, and can keep paying for that.
>
> **Rent:** you use the hop as its owner ships it, and you watch the owner's patch channel and delay.
>
> **Pin:** you hold the hop at a known version for stability. Write the support end date next to it.
>
> **Isolate:** you cannot trust the hop and cannot replace it, so you contain it: a sandbox, a separate process, a separate desktop, no capture tools on the screen.

APPENDIX C

The Five-OS Reference Table

Every hop of the frame on Windows, macOS, iOS, Linux and Android, with the components that do the work. macOS and iOS have their own columns because their app frameworks, capture rules and engine rules differ. Use this table to fill column 1 of the Frame Ledger; the chapters explain each cell. Components and versions are as of November 2026, checked against the vendor documentation listed at the end.

How to read it

- **Web content** means a browser or an embedded web view. On Windows that is usually Blink inside Edge or Microsoft's Edge WebView2 runtime; on macOS and iOS, WebKit inside Safari or `WKWebView`; Linux has no OS web view, so apps use WebKitGTK, Qt WebEngine (Chromium-based) or a bundled Chromium; on Android, Chrome or the Android System WebView, both Chromium.
- **Native** means the platform's own UI framework. Native apps, hardware overlays and direct or independent flip can skip part of the chain, so not every row applies to every screen.
- On iOS, apps must use WebKit for web content unless they hold one of Apple's alternative browser engine entitlements. In the EU (from iOS 17.4) these cover browser apps and apps that offer in-app browsing, distributed only in the EU. In Japan (from iOS 26.2) they cover browser apps, and in-app browsing only for apps from browser engine stewards. Everywhere else, and in apps without an entitlement, embedded web views use WebKit (Chapter Thirteen).

From code to the engine compositor (hops 0 to 7)

Hop	Windows	macOS	iOS	Linux	Android
0 Source	Win32, WPF, WinUI 3 (XAML); web through Edge or WebView2	AppKit, SwiftUI; web through `WKWebView`	UIKit, SwiftUI; web through `WKWebView`	GTK, Qt; web through WebKitGTK, Qt WebEngine or a bundled Chromium	Views, Jetpack Compose; web through the System WebView
1 Parse	Blink HTML parser; XAML parser for native	WebKit (WebCore) parser	WebKit (WebCore) parser	WebCore, Blink or Gecko, by engine	Blink HTML parser; layout XML inflation for Views
2 Style	Blink style engine; XAML styles and templates	WebCore style resolution	WebCore style resolution	WebCore, Blink, or Gecko's Stylo (Rust)	Blink style engine; themes and styles for Views
3 Layout	Blink LayoutNG; XAML measure and arrange	WebCore layout; Auto Layout for AppKit	WebCore layout; Auto Layout for UIKit	Engine layout; GTK and Qt layout managers	Blink LayoutNG; View measure and layout; Compose layout
4 Text	Chromium: HarfBuzz shaping, DirectWrite for fonts, metrics and glyph raster; Fontations (Rust) as the web font fallback where the system cannot read a format (Chrome 133), FreeType removed from Blink (Chrome 145). Native: DirectWrite	Core Text	Core Text	Chromium: HarfBuzz shaping; Fontations (Rust) parses fonts and loads outlines for all web fonts (Chrome 133) and system fonts (Chrome 136), Skia rasterizes; FreeType removed from Blink (Chrome 145). GTK and other native apps: HarfBuzz shaping, FreeType raster, Fontconfig font choice, Pango for GTK text layout	Minikin on HarfBuzz and ICU; glyphs drawn through Skia
5 Paint	Blink display lists; Direct2D and the XAML visual tree for native	WebKit display lists; Core Graphics and Core Animation layer content	WebKit display lists; Core Graphics and Core Animation layer content	Engine display lists; Cairo (GTK 3), GSK render nodes (GTK 4), Qt Quick scene graph	HWUI records display lists from Canvas calls
6 Raster	Skia on the GPU in Chromium's GPU process, through ANGLE on Direct3D 11; Direct2D on Direct3D for native	Core Graphics and Metal	Core Graphics and Metal	Skia or the engine's rasterizer on OpenGL or Vulkan, through Mesa or the vendor driver	HWUI render thread draws with Skia on OpenGL ES or Vulkan

7 Engine compositor	Chromium `cc` (compositor thread) and Viz; Windows composition visual layer for WinUI	Core Animation layer tree	Core Animation layer tree	Chromium Viz, the GTK renderer, the Qt Quick renderer	HWUI render thread; Chromium Viz inside WebView

From the OS compositor to the readers (hop 8 to Capture)

Hop	Windows	macOS	iOS	Linux	Android
8 OS compositor	DWM (`dwm.exe`); always on since Windows 8, though independent flip and multiplane overlays can scan a surface out without DWM drawing it	WindowServer	Core Animation's out-of-process render server, the `backboardd` process	Wayland compositor (Mutter, KWin, wlroots-based, Weston); or Xorg with a compositing manager	SurfaceFlinger, with the Hardware Composer HAL
9a Frame pacing for apps (vsync APIs)	DXGI swap-chain presentation and frame latency waitable objects	`CADisplayLink` (macOS 14 and later)	`CADisplayLink`	Wayland frame callbacks and the presentation-time protocol	`Choreographer` delivers vsync to apps
9b Scanout (overlays, planes, display)	DXGI flip model, independent flip, multiplane overlays; WDDM display driver	Display driver under WindowServer; variable refresh on supported displays	System display pipeline; variable refresh on supported devices	DRM and KMS (atomic mode setting, planes)	Hardware Composer HAL and display planes
10 Glass and readers: assistive APIs	UI Automation	NSAccessibility	UIAccessibility	AT-SPI	AccessibilityService
Input	Pointer and touch messages to the window under the point; the engine hit-tests the page	AppKit events, hit-tested through the view hierarchy	UIKit touches, hit-tested through the view hierarchy	libinput in the compositor; Wayland seat events to the focused surface	InputDispatcher sends `MotionEvent` to the target window
Capture APIs	Windows.Graphics.Capture (Windows 10, version 1803); Desktop Duplication API	ScreenCaptureKit (introduced at WWDC22, available from macOS 12.3), behind the user's screen recording permission	ReplayKit; apps can watch `isCaptured` (iOS 11)	Wayland: xdg-desktop-portal ScreenCast, streams delivered through PipeWire. X11: no capture restriction; any client can read the screen	MediaProjection (API level 21), with user consent
Capture exclusion	`SetWindowDisplayAffinity`; `WDA_EXCLUDEFROMCAPTURE` from Windows 10, version 2004. Microsoft: no guarantee, not a security feature; does not stop a camera	`NSWindow sharingType`; test it against each capture API on each release; does not stop a camera	No per-window exclusion for apps; FairPlay Streaming video is blacked out in captures	No standard exclusion; under Wayland the portal asks the user what to share; under X11 nothing can exclude a window	`FLAG_SECURE`: enforced against supported screenshots and non-secure displays; it does not stop a camera, an AccessibilityService reading the view tree, or a compromised device

Trace tools	Windows Performance Analyzer, PIX; Perfetto or Chrome tracing for Chromium	Xcode Instruments	Xcode Instruments	Perfetto or Chrome tracing; driver and compositor debug tools	Perfetto; `dumpsys gfxinfo`

THREE CELLS THAT CHANGE DECISIONS

The OS compositor row is not optional on Windows. Since Windows 8, DWM composition is always enabled, so DWM manages every windowed Windows screen, and display affinity works only because DWM composes the desktop. A surface on independent flip or a multiplane overlay can reach the display without DWM drawing it into the desktop image, so check the presentation mode before you assume DWM composed a given frame.

The capture exclusion row is a deterrent row. No API in it stops a camera. Microsoft's own page says display affinity gives "no guarantee" that content is protected; Android's `FLAG_SECURE` is enforced, but only against the capture paths the system controls.

The Linux capture cell depends on the session. Under Wayland, capture goes through the portal and the user's choice. X11 has no capture restriction at all: any client can read the whole screen, which is why Wayland plus the portal is a real boundary and an X11 session is not. GNOME 49 (September 2025) disabled its X11 session by default, and GNOME 50 (March 2026) shipped Mutter without its X11 backend, so X11 apps run there only through XWayland; many fleets on other desktops still run X11.

Sources for this table

- Chromium RenderingNG architecture (pipeline stages; render, browser and Viz processes): developer.chrome.com/docs/chromium/renderingng-architecture
- Microsoft Learn: DwmIsCompositionEnabled ("As of Windows 8, DWM composition is always enabled"); SetWindowDisplayAffinity; GraphicsCaptureSession (Windows 10, version 1803); Desktop Duplication API; DXGI flip model
- Apple Developer: ScreenCaptureKit (WWDC22 session 10156, "Meet ScreenCaptureKit"; framework availability macOS 12.3); Core Animation render server and `backboardd` (Apple, "Understanding the visionOS render pipeline"; WWDC14 session 419); UIScreen isCaptured (iOS 11); CADisplayLink (macOS 14 and later); NSWindow sharingType
- Android: WindowManager.LayoutParams FLAG_SECURE; MediaProjection (API level 21); source.android.com SurfaceFlinger and WindowManager; Hardware Composer HAL; Minikin source (HarfBuzz and ICU)

- freedesktop.org: xdg-desktop-portal ScreenCast; GNOME 49 release notes for developers; Mutter NEWS, 50.alpha ("Drop the X11 backend"); Chrome for Developers, "Memory safety for web fonts" (Fontations, Chrome 133 and 145)

Full URLs are in Appendix E.

APPENDIX D

Measurements from the Source

Some claims in this book are not taken from a vendor's page but measured from the code itself: when a component appeared, when it was replaced, how active a project has been. This appendix shows the method, so anyone can repeat it, and lists only the measurements that were actually run. Where a repository was not yet available, the measurement is marked pending; on 18 November 2026 that was only Chromium. Nothing here is estimated.

The method: partial clones and read-only history

The history of a browser engine is large, but its commit history is small next to its files. A **partial clone** downloads the full list of commits and leaves file contents to be fetched only when needed. For the largest projects, a **sparse checkout** then writes only the rendering paths to disk. The result is a library that answers "when was this file added?" without downloading a whole engine.

Step	Command
Clone history, no files yet	`git clone --filter=blob:none --no-checkout <url> <name>`
Limit large repositories to the rendering paths	`git -C <name> sparse-checkout set --no-cone <paths>`
Write the files	`git -C <name> checkout`
First commit of the project	`git log --reverse --format="%H %ad %an | %s" --date=iso-strict | head -1`
When a path was first added	`git log --diff-filter=A --reverse --format="%H %ad %s" --date=short -- <path> | head -1`
When a path was deleted or renamed away	`git log --diff-filter=D --format="%H %ad %s" --date=short -- <path> | head -1`
Commits per year	`git rev-list --count --since=<Y>-01-01 --until=<Y>-12-31 HEAD`

Three cautions apply to every number below. A date of first add is the date a path appeared in this repository; code moved in from elsewhere can be older. A rename shows up as a delete and an add. And commit counts measure activity, not funding, team size or cost: they include merge commits, and one commit can be one line or ten thousand.

Servo: measured

Repository `github.com/servo/servo`, measured at commit `ff0b0187079e8877fdf298b93b25c96223f5a710` (18 November 2026). The history holds 59,387 commits.

What	Commit	Date	Result
First commit	`ce30d4520d`	8 February 2012	"initial add"
Today's layout engine first added (`components/layout_2020`)	`2b01c26aa5`	27 June 2019	Added behind a build flag, "--with-layout-2020"
WebGPU first added (`components/webgpu`)	`12893aa010`	10 November 2019	"Initial implementation of WebGPU API"
Font code renamed (`gfx` to `components/fonts`)	`cd2ab36759`	19 June 2024	Rename, not new code
Legacy layout removed	`7594dc6991`	13 March 2025	"Remove legacy layout (layout 2013)"
New layout becomes the only `layout` crate	`7787cab521`	19 April 2025	`layout_2020` merged into a crate called `layout`
Compositor renamed (`compositing` to `paint`)	`9c9d9c863f`	24 January 2026	Rename, not new code

So the layout engine Servo runs today started as a second engine beside the first in June 2019 and replaced it fully almost six years later, in March 2025. Running two engines side by side for years is the same pattern Blink followed with LayoutNG: the replacement of a renderer hop is measured in years, not releases.

Servo's commits per year, 2012 to 2026, are drawn in Figure 12.1 and are not repeated here. They fall from 3,604 in 2020 to 776 in 2021 and 520 in 2022, the years after Mozilla cut the team in August 2020, and climb back from 2023. That is an activity curve, and it is the

clearest one in this book of what happens to an engine when its main payer leaves. It says nothing on its own about who pays today; Chapter Thirteen covers that from published records.

The workspace manifest (`Cargo.toml` at the same commit) also shows what Servo rents rather than owns: `webrender` 0.70.1 for compositing and raster, `harfbuzz-sys` 0.8 for shaping, and the Rust font crates `read-fonts` 0.46 and `skrifa` 0.49.0. Its release tags: `v0.0.1` on commit `721214fbe4` (19 October 2025), `v0.1.0` on `8032867a0d` (11 April 2026) and `v0.8.0` on `4e136d8ef3` (5 November 2026); the GitHub releases were published on 20 October 2025, 13 April 2026 and 9 November 2026.

The other repositories: first commits

Measured on 18 November 2026 from the clones in the source library, with `git log --reverse` and `git rev-list --count HEAD`. A first commit is the first in this repository's history; projects that moved from older version control may begin with an import.

Repository	First commit	Date	Commits at head (head date)
WebKit (`github.com/WebKit/WebKit`)	`e6896ffd9b1`, "New repository initialized by cvs2svn"	24 August 2001	327,080 (`f31b755f63d`, 17 November 2026)
Firefox monorepo, Gecko inside (`github.com/mozilla-firefox/firefox`)	`c4cc52826abd`, "Free the lizard"	28 March 1998	1,013,399 (`524e4e57a3`, 18 November 2026)
Skia	`586101c79b`, "Initial directory structure."	20 September 2006	81,869 (`90f942bba5`, 17 November 2026)
HarfBuzz	`80a158291`	20 December 2000	22,640 (`7ed6825ed`, 17 November 2026)
FreeType	`d2b1f357`, "Initial revision"	16 December 1999	8,681 (`d7339729`, 17 November 2026)
CEF	`11f5769f`, "Initial directory structure."	26 November 2008	3,746 (`e6035cfb`, 12 November 2026)
Electron	`e451d9212`, "Initial commit"	13 March 2013	31,559 (`97476cbe6`, 17 November 2026)

Android `platform/frameworks/native`	`7c1b96a165`, "Initial Contribution"	21 October 2008	126,370 (`4f463a6b1d`, 25 March 2025: the public branch has had no commits since)
Weston	`97f1ebe8`, "Initial commit."	30 September 2008	11,544 (`749e4077`, 17 November 2026)
wlroots	`52e6ed54`, "Initial commit"	25 April 2017	7,958 (`df787883`, 5 November 2026)
Vello	`1b0248fb`, "Starting piet-gpu repo"	5 April 2020	2,151 (`84ffc24a`, 16 November 2026)
wry	`3e326e3`, "Init commit"	12 July 2020	1,221 (`89c4957`, 18 November 2026)

The firefox history also holds a few commits whose author date is 1 January 1970, a known artifact of converted history; they are ignored above, and the first commit is the earliest real date.

Key paths: when each hop's code appeared

Measured with `git log --diff-filter=A --reverse -- <path>`. Where the first add is a move or rename, the table says so; the code is older than that date.

Hop	Repository and path	First added	What the commit says
1 to 5 Engine	WebKit `WebCore/khtml`	24 August 2001 (`a063c2b75dd`)	"Initial revision": the KHTML import Apple's engine descends from
3 Layout	WebKit `Source/WebCore/layout`	23 April 2018 (`c1cd21d32b7`)	"[LayoutFormattingContext] Initial commit"
1 to 7 Process model	WebKit `WebKit2`	8 April 2010 (`a0a92404f15`)	The multi-process WebKit2 tree; moved to `Source/WebKit2` on 15 January 2011
6 to 7 GPU process	WebKit `Source/WebKit/GPUProcess`	4 December 2019 (`2d9d0c575ad`)	"Introduce a GPU process"

2 Style	firefox `layout/style/quirk.css`	2 June 2000 (`a492e0b204da`)	"QuirkMode style sheet"; today at `layout/style/res/quirk.css`
2 Style	WebKit `WebCore/khtml/css/quirks.css`	25 February 2003 (`9d6c8afab70`)	Today at `Source/WebCore/css/quirks.css`
7 Engine compositor	firefox `gfx/layers`	1 March 2010 (`46c50d3967fb`)	"Create initial layers API"
6 to 7 Raster and compositor	firefox `gfx/webrender`, then `gfx/wr`	6 February 2017 (`52c4335d06f9`); moved 22 November 2018 (`845b81ddb8c7`)	WebRender's Rust crates brought into the Firefox tree
1 to 7 Sandbox	firefox `security/sandbox`	12 August 2013 (`59aaccaaa1b6`)	seccomp-bpf sandboxing for B2G
6 Raster (image decode)	WebKit `WebCore/platform/image-decoders/webp/WEBPImageDecoder.cpp`	18 October 2010 (`20d7e999518`)	WebP decoder wrapping libwebp (C)
6 Raster (image decode)	firefox `image/decoders/nsWebPDecoder.cpp`	3 October 2018 (`ea3496614c4a`)	"Implement WebP decoder"
5 to 6 Paint and raster	Skia `src/core/SkCanvas.cpp`	17 December 2008 (`8a1c16ff38`)	"grab from latest android"
6 Raster	Skia `src/gpu/graphite`	8 April 2022 (`ae5e846047`)	Graphite moved into Skia's base directories
4 Text	Skia `src/ports/fontations`	6 April 2023 (`97113b7391`)	Fontations Rust crates added
6 Raster (image decode)	Skia `experimental/rust_png`, then `rust/png`	9 August 2024 (`cbd5898acb`); moved 17 September 2025 (`3745e62ff9`)	"Initial implementation of `SkPngRustCodec`"

4 Text	FreeType `src/truetype/ttinterp.c`	16 December 1999 (`d2b1f357`)	In the first commit: the TrueType bytecode interpreter
4 Text	HarfBuzz `src/hb-ot-layout.cc`	24 January 2008 (`fd92a3dde`)	"Starting public interface"
1 to 7 Embedded engine	CEF `libcef`	2 December 2008 (`9c58b671`)	"Initial commit of libcef and cefclient projects"
1 to 7 Embedded engine	Electron `atom`, renamed `shell`	16 March 2014 (`516d46444`); renamed 19 June 2019 (`d7f07e8a8`)	Move, then rename; not new code
8 OS compositor	Android `services/surfaceflinger/SurfaceFlinger.cpp`	14 July 2010 (`81bac09fa6`)	"move native services under services/": a move, the code is older
8 OS compositor	Android `services/surfaceflinger/CompositionEngine`	18 October 2018 (`70d9136897`)	"SF: Introduce a minimal CompositionEngine"
8 OS compositor	wlroots `backend/drm`	1 May 2017 (`aca13320`)	From the project's first weeks
6 Raster	Vello `sparse_strips`	28 February 2025 (`1c312e3e`)	The CPU and hybrid sparse-strip renderers

Two patterns repeat. Old hops stay: FreeType's interpreter, the KHTML-descended WebCore and both quirks stylesheets are still in their trees. And replacements arrive beside the old code, years before they take over: WebKit's new layout formatting contexts (2018), Firefox's WebRender (2017 in the tree, 2019 rollout) and Skia's Rust PNG decoder (2024) all started next to the code they were built to replace.

Still pending

Repository	Paths to measure	Status on 18 November 2026
Chromium (`chromium/src`)	`cc/`, `components/viz/`, Blink `core/layout/` and `core/paint/`, `platform/graphics/`, `gpu/`, `sandbox/`	Clone still downloading; no path dates claimed. The 2025 commit-domain count in Chapter Thirteen used a separate treeless clone, recorded in its source notes

Each measurement a chapter cites is recorded, with the exact command and its output, in that chapter's source notes, and is gathered into this appendix for the final edition.

APPENDIX E

Reading List and Sources

Part 1 is a short shelf of primary sources behind the whole book: standards, vendor documentation, engine design documents, court records and original papers. Part 2 lists the sources each chapter relied on, taken from the source notes kept with every chapter, so a reader can check any claim at its origin. Sources were checked in November 2026; web pages change, so check the date on anything you rely on.

Part 1: Primary sources for the whole book

Source	Why read it	Where
Chromium, RenderingNG architecture	The pipeline stages and processes of the engine behind Chrome, Edge, WebView2 and Electron.	`https://developer.chrome.com/docs/chromium/renderingng-architecture`
Chromium, LayoutNG	How Blink replaced its legacy layout engine.	`https://developer.chrome.com/docs/chromium/layoutng`
Chromium blog, "Blink: A rendering engine for the Chromium project" (April 2013)	The fork from WebKit.	`https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html`
Chromium, Multi-process architecture	Why the browser splits into browser, renderer and GPU processes.	`https://www.chromium.org/developers/design-documents/multi-process-architecture/`
Chromium, The Rule of 2	Chromium's rule for code that handles untrusted input.	`https://chromium.googlesource.com/chromium/src/+/main/docs/security/rule-of-2.md`
Google Security Blog, "Mitigating Spectre with Site Isolation in Chrome" (July 2018)	Site Isolation and its memory cost.	`https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html`
WHATWG HTML Standard, Parsing HTML documents	The parse hop, as the standard defines it.	`https://html.spec.whatwg.org/multipage/parsing.html`
W3C, CSS Fonts Module Level 4	How fonts are chosen and loaded, including remote fonts.	`https://www.w3.org/TR/css-fonts-4/`

HarfBuzz manual	The shaping engine used by Chromium, Android and Linux desktops.	https://harfbuzz.github.io/
FreeType project	The font rasterizer used on Linux, Android and in many apps.	https://freetype.org/
Skia	The 2D graphics library behind Chrome and Android.	https://skia.org/
Microsoft Learn, DirectWrite	The Windows text API.	https://learn.microsoft.com/en-us/windows/win32/directwrite/direct-write-portal
Apple Developer, Core Text	The text stack of macOS and iOS.	https://developer.apple.com/documentation/coretext
Microsoft Learn, Desktop Window Manager and DwmIsCompositionEnabled	Desktop composition since Vista; always on since Windows 8.	https://learn.microsoft.com/en-us/windows/win32/api/dwmapi/nf-dwmapi-dwmiscompositionenabled
Microsoft Learn, DXGI flip model	Flip presentation, independent flip and its effect on composition.	https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/dxgi-flip-model
Apple Developer, Core Animation (QuartzCore) and CADisplayLink	Layers, the render server and frame pacing on Apple platforms.	https://developer.apple.com/documentation/quartzcore
Android Open Source Project, SurfaceFlinger and WindowManager	Android's compositor and its use of the Hardware Composer.	https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager
Android Open Source Project, Hardware Composer HAL	Display planes and composition on Android.	https://source.android.com/docs/core/graphics/hwc
Wayland project	The Linux display protocol.	https://wayland.freedesktop.org/
Linux kernel documentation, KMS	Kernel mode setting and display planes.	https://docs.kernel.org/gpu/drm-kms.html
GNOME 49, what's new for developers	The X11 session disabled by default.	https://release.gnome.org/49/developers/

Microsoft Learn, SetWindowDisplayAffinity	Display affinity and its stated limits.	https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowdisplayaffinity
Microsoft Learn, GraphicsCaptureSession	Windows.Graphics.Capture, from Windows 10, version 1803.	https://learn.microsoft.com/en-us/uwp/api/windows.graphics.capture.graphicscapturesession
Apple, WWDC23 session 10136, ScreenCaptureKit	Introduced in macOS 12.3.	https://developer.apple.com/wwdc23/10136
Apple Developer, UIScreen isCaptured	How an iOS app learns the screen is being captured.	https://developer.apple.com/documentation/uikit/uiscreen/iscaptured
Android Developers, WindowManager.LayoutParams FLAG_SECURE	The secure-window flag and what it covers.	https://developer.android.com/reference/android/view/WindowManager.LayoutParams#FLAG_SECURE
Android Developers, MediaProjection	Screen capture on Android, from API level 21.	https://developer.android.com/reference/android/media/projection/MediaProjection
Android Developers, Android Protected Confirmation	Hardware-protected confirmation and what it proves.	https://developer.android.com/privacy-and-security/security-android-protected-confirmation
xdg-desktop-portal, ScreenCast	Screen capture on Wayland desktops through PipeWire.	https://flatpak.github.io/xdg-desktop-portal/docs/doc-org.freedesktop.portal.ScreenCast.html
Microsoft Learn, UI Automation	The Windows accessibility API.	https://learn.microsoft.com/en-us/windows/win32/winauto/entry-uiauto-win32
Microsoft Learn, WebView2 release notes	The Edge runtime for apps and its update channels.	https://learn.microsoft.com/en-us/microsoft-edge/webview2/release-notes/
Electron, release timelines	Electron's majors and support window.	https://www.electronjs.org/docs/latest/tutorial/electron-timelines
Chromium Embedded Framework	The CEF project.	https://bitbucket.org/chromiumembedded/cef
Tauri	Apps on the OS web view.	https://v2.tauri.app/

Qt, Qt WebEngine overview	The Chromium-based web engine in Qt.	https://doc.qt.io/qt-6/qtwebengine-overview.html
Android Developers, WebView	The Android System WebView.	https://developer.android.com/develop/ui/views/layout/webapps/webview
Apple, App Review Guidelines and alternative browser engines	When iOS apps must use WebKit, and the EU and Japan exceptions.	https://developer.apple.com/support/alternative-browser-engines/
Servo	The Rust browser engine measured in Appendix D.	https://servo.org/
Mozilla, Firefox 67 release notes	WebRender begins its rollout.	https://www.mozilla.org/en-US/firefox/67.0/releasenotes/
Delegated Regulation (EU) 2018/389 (PSD2 RTS)	Article 5, dynamic linking of amount and payee.	https://eur-lex.europa.eu/eli/reg_del/2018/389/oj
US Department of Justice, U.S. and Plaintiff States v. Google LLC	The search case record and opinions.	https://www.justice.gov/atr/case/us-and-plaintiff-states-v-google-llc
CISA, Known Exploited Vulnerabilities Catalog	Which rendering CVEs were exploited in the wild.	https://www.cisa.gov/known-exploited-vulnerabilities-catalog
Citizen Lab, FORCEDENTRY (September 2021) and BLASTPASS (September 2023)	Zero-click exploits through image and font parsing.	https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/
Project Zero, "A deep dive into an NSO zero-click iMessage exploit" (December 2021)	How FORCEDENTRY worked.	https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html
GPU.zip	A pixel-stealing side channel through GPU data compression.	https://www.hertzbleed.com/gpu.zip/
A. Ng, J. Lepinski, D. Wigdor, S. Sanders, P. Dietz, "Designing for Low-Latency Direct-Touch Input", UIST 2012	How small a touch delay people can notice.	https://doi.org/10.1145/2380116.2380174

R. Jota, A. Ng, P. Dietz, D. Wigdor, "How Fast is Fast Enough? A Study of the Effects of Latency in Direct-Touch Pointing Tasks", CHI 2013	Latency and task performance.	https://doi.org/10.1145/2470654.2481317
Perfetto	The tracing tool for Chromium and Android frames.	https://perfetto.dev/

Part 2: Sources by chapter

Reader's Brief

- WebView2 Evergreen updates automatically and receives the same updates as Microsoft Edge; one shared copy; hard-linked with Edge on eligible systems: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/distribution
- WebView2 Evergreen updates automatically and receives the same updates as Microsoft Edge; one shared copy; hard-linked with Edge on eligible systems: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/evergreen-vs-fixed-version
- Microsoft's performance page: each WebView2 control creates its own set of processes (browser, renderer, GPU); the process-model page gives the fuller picture (see below): https://learn.microsoft.com/microsoft-edge/webview2/concepts/performance
- Chromium renderers run at Untrusted integrity, the GPU process at Low (as of M75): https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md
- DWM cannot be programmatically disabled as of Windows 8 ("has no effect"): https://learn.microsoft.com/en-us/windows/win32/api/dwmapi/nf-dwmapi-dwmenablecomposition
- SetWindowDisplayAffinity: "no guarantee ... will strictly protect windowed content, for example where someone takes a photograph of the screen": https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowdisplayaffinity
- PSD2 RTS Art. 5(1)(a): "the payer is made aware of the amount of the payment transaction and of the payee": https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32018R0389
- Integrity levels: "a principal with a low integrity level cannot write to an object with a medium integrity level"; standard users receive medium; Untrusted is a defined ...: https://learn.microsoft.com/en-us/windows/win32/secauthz/mandatory-integrity-control
- Integrity levels: "a principal with a low integrity level cannot write to an object with a medium integrity level"; standard users receive medium; Untrusted is a defined ...: https://learn.microsoft.com/en-us/windows/win32/secauthz/well-known-sids

- Process sharing: one browser process per user data folder; helper processes (GPU) in the same group; a renderer can serve frames in several controls: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/process-model
- KEV catalog as the trigger for the sample clause's shorter tier: https://www.cisa.gov/known-exploited-vulnerabilities-catalog
- Hops 3 and 4 run together: Blink's ShapingLineBreaker "Shapes a line of text by finding the ideal break position as indicated by the available space and the shape ...: https://chromium.googlesource.com/chromium/src/+/HEAD/third_party/blink/renderer/platform/fonts/shaping/shaping_line_breaker.h
- Viz: Chromium's display compositor that assembles the frame in the GPU process: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/how_cc_works.md
- Viz: Chromium's display compositor that assembles the frame in the GPU process: https://developer.chrome.com/docs/chromium/renderingng-data-structures
- DWM row: independent flip and MPO can take a frame past DWM composition: https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/for-best-performance--use-dxgi-flip-model

Prologue: One Tap, One Frame

- WM_POINTERDOWN, minimum client Windows 8: https://learn.microsoft.com/en-us/windows/win32/inputmsg/wm-pointerdown
- A message loop in each thread that creates a window: https://learn.microsoft.com/en-us/windows/win32/winmsg/about-messages-and-message-queues
- Windowed hosting: "The OS sends the input to the WebView2"; Visual hosting forwards input; DirectComposition visuals: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/windowed-vs-visual-hosting
- Runtime executable msedgewebview2.exe: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/distribution
- Microsoft's performance page: each WebView2 control "creates its own set of processes, such as browser, renderer, and GPU" (a simplification; see the process-model row ...: https://learn.microsoft.com/microsoft-edge/webview2/concepts/performance
- Renderers Untrusted, GPU process Low integrity (M75): https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md
- Commit blocks the main thread while data is copied; renderer uses multi-threaded compositor: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/how_cc_works.md
- DWM: windows "no longer draw directly to the screen" (Vista); disabling "has no effect" as of Windows 8: https://learn.microsoft.com/en-us/windows/win32/dwm/dwm-overview

- DWM: windows "no longer draw directly to the screen" (Vista); disabling "has no effect" as of Windows 8: https://learn.microsoft.com/en-us/windows/win32/api/dwmapi/nf-dwmapi-dwmenablecomposition
- Flip presentation and overlays can bypass composition: https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/for-best-performance--use-dxgi-flip-model
- Display affinity quote: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowdisplayaffinity
- iOS alternative engines: EU entitlements on iOS 17.4 and later; Japan on iOS 26.2 and later: https://developer.apple.com/support/alternative-browser-engines/
- iOS alternative engines: EU entitlements on iOS 17.4 and later; Japan on iOS 26.2 and later: https://developer.apple.com/support/alternative-browser-engines-jp/
- Wayland frame callback: "Request a notification when it is a good time to start drawing a new frame": https://gitlab.freedesktop.org/wayland/wayland/-/raw/main/protocol/wayland.xml
- Android: SurfaceFlinger composites on each VSync: https://source.android.com/docs/core/graphics/implement-vsync
- Process model: "a single browser process" per WebView2 process group; one group per user data folder; GPU process is a helper in the group; a renderer "can serve frames ...: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/process-model
- Hops 3 and 4 run together (Blink shaping_line_breaker.h): https://chromium.googlesource.com/chromium/src/+/HEAD/third_party/blink/renderer/platform/fonts/shaping/shaping_line_breaker.h

Chapter 1. Sixteen Milliseconds: The Frame Budget and the Ledger

- Profile GPU Rendering: green line at 16.67 ms; needs Android 4.1 (API level 16): https://developer.android.com/topic/performance/rendering/inspect-gpu-rendering
- ProMotion 10 to 120 Hz on supported iPhones, 24 to 120 Hz on iPad Pro; "your app can't presume a specific refresh rate at any time": https://developer.apple.com/documentation/quartzcore/optimizing-iphone-and-ipad-apps-to-support-promotion-displays
- Windows 11 Dynamic Refresh Rate "automatically adjusts the refresh rate based on what you are doing at the time"; needs VRR and at least 120 Hz: https://support.microsoft.com/help/4583945
- Win32 message loop per thread that creates a window; GetMessage, TranslateMessage, DispatchMessage: https://learn.microsoft.com/en-us/windows/win32/winmsg/about-messages-and-message-queues
- "the Android UI toolkit is not thread-safe. So, don't manipulate your UI from a worker thread.": https://developer.android.com/guide/components/processes-and-threads

- cc scheduler "maintains a deadline by which it expects its embedder to respond"; BeginImplFrame "driven by a viz::BeginFrameSource", driven by the display compositor; ...: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/how_cc_works.md
- Frames N, N+1, N+2; "The latency is at least two frames"; VSync offsets: https://source.android.com/docs/core/graphics/implement-vsync
- DXGI: "almost a full extra frame"; waitable swap chain from Direct3D 11.2 / DXGI 1.3; default frame latency 1: https://learn.microsoft.com/en-us/windows/uwp/gaming/reduce-latency-with-dxgi-1-3-swap-chains
- Ng, Lepinski, Wigdor, Sanders, Dietz, UIST 2012 (pp. 453 to 462): 1 ms baseline prototype; dragging, users unable to perceive below about 6 ms, range 2 to 11 ms. Jota, ...: https://sites.ualberta.ca/~wfb/publications/C-2014-SIGCHI-Latency.pdf
- WPR built-in profiles "GPU activity" and "Desktop composition activity" ("All desktop composition activity"): https://learn.microsoft.com/en-us/windows-hardware/test/wpt/built-in-recording-profiles
- WebView2.wprp ETW profile; Edge DevTools via Inspect and edge://inspect: https://learn.microsoft.com/microsoft-edge/webview2/concepts/performance
- PresentMon: Intel, MIT license, ETW-based CPU, GPU and display frame durations and latencies: https://github.com/GameTechDev/PresentMon
- Perfetto: "the default tracing system for the Android OS and the Chrome Browser"; Linux probes; ui.perfetto.dev opens Chrome JSON traces: https://perfetto.dev/docs/
- FrameTimeline: SurfaceFlinger module, Android 12 or higher; AppDeadlineMissed, SurfaceFlinger CPU/GPU deadline missed; frames "presented late, resulting in an increased ...: https://perfetto.dev/docs/data-sources/frametimeline
- gfxinfo framestats: last 120 frames, nanosecond timestamps, from Android 6.0 (M): https://webarchive.library.unt.edu/web/20160706090017mp_/https://developer.android.com/training/testing/performance.html
- Instruments Animation Hitches; hitch = frame late to screen; hitch time ratio in ms per s; under 5 ms/s good; MetricKit hitch metric: https://developer.apple.com/videos/play/wwdc2020/10077
- Instruments Animation Hitches; hitch = frame late to screen; hitch time ratio in ms per s; under 5 ms/s good; MetricKit hitch metric: https://developer.apple.com/documentation/metrickit/hitchtimemetric
- CADisplayLink: iOS 3.1, macOS 14.0: https://developer.apple.com/tutorials/data/documentation/quartzcore/cadisplaylink.json
- Choreographer: added in API level 16; "Coordinates the timing of animations, input and drawing": https://developer.android.com/reference/android/view/Choreographer

- wl_surface.frame request text: https://gitlab.freedesktop.org/wayland/wayland/-/raw/main/protocol/wayland.xml
- Windows 1.0 released 20 November 1985: https://en.wikipedia.org/wiki/Windows_1.0
- Windows 1.0 released 20 November 1985: https://www.computinghistory.org.uk/det/5406/Microsoft-releases-Windows-1-0/
- Chrome four-week cycle from Chrome 94 (21 Sept 2021): https://developer.chrome.com/blog/faster-release-cycle/
- WebView2 Fixed Version: most-patched builds of the latest and second-latest majors offered: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/distribution
- Renderer Untrusted, GPU process Low integrity: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md
- Jota, Ng, Dietz, Wigdor, "How Fast is Fast Enough? A Study of the Effects of Latency in Direct-Touch Pointing Tasks", CHI 2013, pp. 2291 to 2300: 1 ms reference; ...: https://www.tactuallabs.com/papers/howFastIsFastEnoughCHI13.pdf
- Ng, Lepinski, Wigdor, Sanders, Dietz, "Designing for Low-Latency Direct-Touch Input", UIST 2012 (DOI 10.1145/2380116.2380174): 1 ms prototype, dragging; floor about 6 ms ...: https://doi.org/10.1145/2380116.2380174
- iPhone 14 Pro (press release 7 Sept 2022): ProMotion "features the Always-On display for the first time ever on iPhone, enabled by a new 1Hz refresh rate": https://www.apple.com/newsroom/2022/09/apple-debuts-iphone-14-pro-and-iphone-14-pro-max/
- Fixed Version: "The most-patched version of the latest and second-latest major releases are available for download at this site." (stated in the distribution guide; the ...: https://developer.microsoft.com/en-us/microsoft-edge/webview2/

Chapter 2. From Markup to Geometry: Parse, Style and Layout

- HTML input "is passed through a tokenization stage followed by a tree construction stage"; "The error handling for parse errors is well-defined": https://html.spec.whatwg.org/multipage/parsing.html
- CSS1 W3C Recommendation 17 Dec 1996: https://www.w3.org/Press/CSS1-REC-PR.html
- CSS1 W3C Recommendation 17 Dec 1996: https://www.w3.org/TR/REC-CSS1
- Firefox 57: "Gecko makes use of the parallel style system written in Rust that comes from Servo": https://firefox-source-docs.mozilla.org/layout/StyleSystemOverview.html
- Forced synchronous layout, layout thrashing, offsetHeight example; DevTools "Forced Reflow" insight; LoAF forcedStyleAndLayoutDuration: https://web.dev/articles/avoid-large-complex-layouts-and-layout-thrashing

- "As of Chrome 108, the legacy engine is no longer used to perform layout"; LayoutNG first parts 2019; fragmentation 102, flex/grid 103, tables 106, printing 108: https://developer.chrome.com/docs/chromium/renderingng-fragmentation
- Safari announced 7 Jan 2003, engine "based on KHTML, from KDE's Konqueror open source project": https://www.apple.com/newsroom/2003/01/07Apple-Unveils-Safari/
- WebKit CVS and bug database opened June 2005: https://dot.kde.org/2005/06/07/apple-opens-webkit-cvs-and-bug-database/
- Blink announced 3 April 2013: https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html
- WebKit Layout Formatting Context code under Source/WebCore/layout, integrated step by step with legacy rendering: https://bugs.webkit.org/show_bug.cgi?id=206526
- Servo Layout 2020 "designed to fix several shortcomings in Layout 2013"; "the best layout engine for Servo going forward" (13 Apr 2023): https://servo.org/blog/2023/04/13/layout-2013-vs-2020/
- Microsoft Edge Legacy support ended 9 March 2021: https://learn.microsoft.com/en-us/lifecycle/products/microsoft-edge-legacy
- Microsoft Edge Legacy support ended 9 March 2021: https://blogs.windows.com/msedgedev/2021/03/09/microsoft-edge-legacy-end-of-support/
- "Three trees" is a simplified model; Blink also keeps an immutable fragment tree, property trees (transform, clip, effect, scroll), display lists and paint chunks: https://developer.chrome.com/docs/chromium/renderingng-data-structures
- Script blocking: "Scripts without async, defer or type=module attributes, as well as inline scripts without the type=module attribute, are fetched and executed ...: https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/script
- ShapingLineBreaker: break from paragraph shape results; reshape unless at a HarfBuzz safe-to-break boundary: https://chromium.googlesource.com/chromium/src/+/HEAD/third_party/blink/renderer/platform/fonts/shaping/shaping_line_breaker.h
- iOS: WebKit required except EU and Japan alternative-engine entitlements: https://developer.apple.com/support/alternative-browser-engines/

Chapter 3. Text: Shaping, Fonts and the Glyph

- Blink shaping implemented in shaping/harfbuzz_shaper.cc (one implementation; platform font caches are separate per OS): https://chromium.googlesource.com/chromium/src/+/main/third_party/blink/renderer/platform/fonts/README.md
- "The sandbox prevents the renderer from using the system font collection"; fallback by IPC to the browser: https://www.chromium.org/developers/design-documents/directwrite-font-proxy/

- Minikin uses HarfBuzz (hb-ot.h, libharfbuzz_ng): https://android.googlesource.com/platform/frameworks/minikin/
- TrueType instructions are a stack-based instruction set: https://learn.microsoft.com/typography/opentype/spec/tt_instructions
- CVE-2011-3402: "TrueType font parsing engine in win32k.sys ... crafted font data in a Word document or web page", exploited in the wild Nov 2011 by Duqu; CISA KEV ...: https://services.nvd.nist.gov/rest/json/cves/2.0?cveId=CVE-2011-3402
- CVE-2011-3402: "TrueType font parsing engine in win32k.sys ... crafted font data in a Word document or web page", exploited in the wild Nov 2011 by Duqu; CISA KEV ...: https://www.cisa.gov/sites/default/files/feeds/known_exploited_vulnerabilities.json
- "With Windows 10, GDI font parsing is no longer performed in kernel mode. Instead, it is performed in a sandboxed user-mode process, fontdrvhost.exe, which executes in a ...: https://techcommunity.microsoft.com/blog/microsoft-security-baselines/dropping-the-untrusted-font-blocking-setting/701068
- Windows 10 and 1607: fontdrvhost.exe but "Installed fonts are still processed in kernel mode"; Windows 10 1703: "All fonts are processed in fontdrvhost.exe in user mode ...: https://msrc.microsoft.com/update-guide/vulnerability/ADV200006
- Untrusted Font Blocking covers GDI fonts outside %windir%\Fonts: https://learn.microsoft.com/windows/security/operating-system-security/device-management/block-untrusted-fonts-in-enterprise
- Fontations: Chrome 128 (Aug 2024) color fonts and CFF2; Chrome 133 (Feb 2025) all web fonts on Linux, Android, ChromeOS, fallback on Windows and Mac; update note: system ...: https://developer.chrome.com/blog/memory-safety-fonts
- "macOS Mojave disabled text subpixel antialiasing by default": https://trac.webkit.org/r239306
- UAX #9 version 17.0.0, 2025-08-13: https://www.unicode.org/reports/tr9/
- UTR #36 stabilized, not maintained since 2014; superseded in part by UTS #39 "Unicode Security Mechanisms" and UTS #55 "Unicode Source Code Handling": https://www.unicode.org/reports/tr36/
- fsType values (Restricted License, Preview & Print, Editable); "Embedding licensing rights are granted by the vendor of the font": https://learn.microsoft.com/typography/opentype/spec/os2
- Monotype "App License" for embedding a font file in a mobile application's code, separate from desktop licensing: https://foundrysupport.monotype.com/hc/en-us/articles/10840068991636-App-License

- TrueType in System 7, May 1991; Windows 3.1, April 1992: https://learn.microsoft.com/typography/truetype/history
- FreeType created 1996 by David Turner; 2.0 (modular rewrite) in 2000: https://freetype.org/history.html
- UAX #9 revision 51 = Version Unicode 17.0.0, dated 2025-08-13; Unicode 17.0.0 released 2025 September 9 (no significant UAX #9 changes); current revision 52 = Version …: https://www.unicode.org/reports/tr9/tr9-51.html
- UAX #9 revision 51 = Version Unicode 17.0.0, dated 2025-08-13; Unicode 17.0.0 released 2025 September 9 (no significant UAX #9 changes); current revision 52 = Version …: https://www.unicode.org/versions/Unicode17.0.0/
- ADV200006 ("Type 1 Font Parsing Remote Code Execution Vulnerability") mitigation table: Windows 10 and 1607 "Installed fonts are still processed in kernel mode"; 1703 …: https://api.msrc.microsoft.com/sug/v2.0/en-US/vulnerability?$filter=cveNumber
- Android: "native code is as sandboxed as interpreted code"; each app gets a unique UID and its own process: https://source.android.com/docs/security/app-sandbox

Chapter 4. Paint and Raster: From Display List to Pixels

- PaintRecord is "the SkPicture equivalent that stores a number of PaintOps"; tiles "roughly 256x256 px" (software raster) and "roughly viewport width x one quarter …: https://chromium.googlesource.com/chromium/src/+/main/docs/how_cc_works.md
- Skia Graphite launched in Chrome on Apple Silicon Macs (8 Jul 2025); "almost 15%" on MotionMark 1.3 on a MacBook Pro M3; Dawn as abstraction over Metal, Vulkan, D3D; …: https://blog.google/chromium/introducing-skia-graphite-chromes/
- Skia project page lists Chrome, ChromeOS, Android, Firefox: https://skia.googlesource.com/skia/+/cbc060a/site/index.md
- Skia Inc. started by Michael Reed in Chapel Hill NC, acquired by Google in 2005; open-sourced with the Chrome release; graphics core of Android and Chrome: https://om.co/gigaom/google-open-sources-skia-graphics-engine/
- Firefox 67 (21 May 2019): "WebRender is gradually enabled by default on Windows 10 desktops with NVIDIA graphics cards": https://www.firefox.com/en-US/firefox/67.0/releasenotes/
- Vello: GPU compute-centric 2D renderer in Rust, alpha: https://github.com/linebender/vello
- HWUI records drawing, RenderThread replays with Skia: https://android.googlesource.com/platform/frameworks/base/+/android16-release/libs/hwui/renderthread/
- "WARP is a high speed, fully conformant software rasterizer": https://learn.microsoft.com/windows/win32/direct3darticles/directx-warp

- "The Gallium LLVMpipe driver is a software rasterizer that uses LLVM to do runtime code generation": https://docs.mesa3d.org/drivers/llvmpipe.html
- Intent to Remove: SwiftShader Fallback, 13 Feb 2025; "approximately 2.7% of WebGL contexts"; "Shipping on Desktop 137"; WebGL creation to fail instead of falling back; ...: https://groups.google.com/a/chromium.org/g/blink-dev/c/yhFguWS_3pM
- DPI-unaware apps render at 96 DPI and are bitmap-stretched, "appearing blurry": https://learn.microsoft.com/windows/win32/hidpi/high-dpi-desktop-application-development-on-windows
- CVE-2023-4863: libwebp heap buffer overflow, Chrome prior to 116.0.5845.187 and libwebp 1.3.2, "crafted HTML page"; KEV dateAdded 2023-09-13: https://services.nvd.nist.gov/rest/json/cves/2.0?cveId=CVE-2023-4863
- BLASTPASS: Citizen Lab, 7 Sep 2023; PassKit attachments containing malicious images via iMessage; Pegasus. CVE-2023-41064 (ImageIO): "Processing a maliciously crafted ...: https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/
- Porter and Duff, "Compositing Digital Images", Computer Graphics 18(3), July 1984 (SIGGRAPH 84), pp. 253 to 259: https://graphics.pixar.com/library/Compositing/paper.pdf
- Graphite on Apple Silicon: Chromium blog "Introducing Skia Graphite: Chrome's rasterization backend for the future", 8 July 2025; "almost 15%" on MotionMark 1.3, MacBook ...: https://blog.chromium.org/2025/07/
- WARP: "a high speed, fully conformant software rasterizer"; renders "where hardware implementations are undesirable or unavailable": https://learn.microsoft.com/en-us/windows/win32/direct3darticles/directx-warp
- SwiftShader is a Google-maintained CPU implementation of graphics APIs: https://github.com/google/swiftshader
- Qt Widgets: raster engine "is used by default for QWidget and QPixmap", "implements all rendering in pure software": https://doc.qt.io/qt-6/qpainter.html
- Qt Quick scene graph "always renders via ... the Qt Rendering Hardware Interface (RHI)" (D3D11, Metal, OpenGL, Vulkan); no Skia: https://doc.qt.io/qt-6/qtquick-visualcanvas-scenegraph-renderer.html
- curl " | base64 -d: https://chromium.googlesource.com/chromium/src/+/main/gpu/config/software_rendering_list.json?format=TEXT"
- curl " | base64 -d: https://chromium.googlesource.com/chromium/src/+/main/gpu/config/gpu_driver_bug_list.json?format=TEXT"

- curl " | base64 -d: https://skia.googlesource.com/skia/+/main/include/core/SkBlendMode.h?format=TEXT"

Chapter 5. Two Compositors, One Vsync

- cc runs in browser and renderer processes; renderer multi-threaded mode "incurs a latency cost"; frames submitted to viz; SurfaceAggregator aggregates frames from all …: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/how_cc_works.md
- WebKit RemoteLayerTree code (RemoteLayerTreeHost, RemoteLayerTreeNode) in Source/WebKit/UIProcess/RemoteLayerTree: https://github.com/WebKit/WebKit/tree/main/Source/WebKit/UIProcess/RemoteLayerTree
- Firefox 67 (21 May 2019) began WebRender rollout to a subset of Windows 10 desktops with NVIDIA GPUs: https://www.ghacks.net/2019/05/21/firefox-67-0-release-information/
- Firefox 67 (21 May 2019) began WebRender rollout to a subset of Windows 10 desktops with NVIDIA GPUs: https://bugzilla.mozilla.org/show_bug.cgi?id=1526061
- DwmEnableComposition deprecated as of Windows 8; "DWM can no longer be programmatically disabled"; disable call returns success with no effect: https://learn.microsoft.com/en-us/windows/win32/api/dwmapi/nf-dwmapi-dwmenablecomposition
- Flip model: "minimal copies"; FLIPEX in D3D9 (Windows 7), FLIP_SEQUENTIAL in DXGI (Windows 8); DirectFlip, panel fitters, MPO; "the DWM can go to sleep"; independent …: https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/for-best-performance--use-dxgi-flip-model
- Multiplane overlays introduced in Windows 8.1 (WDDM 1.3), optional for drivers: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/multiplane-overlay-support
- Vista DWM renders windows to off-screen buffers and composes the desktop: https://learn.microsoft.com/en-us/archive/blogs/greg_schechter/under-the-hood-of-the-desktop-window-manager
- SurfaceFlinger: WindowManager requests layer; BufferQueue passes buffers by handle; on VSYNC walks layers; asks HWC: https://source.android.com/docs/core/graphics/arch-sf-hwc
- HWC marks layers overlay vs client; "Android devices typically support four overlay planes"; overlays "may be less efficient than GL composition when nothing on the …: https://source.android.com/docs/core/graphics/hwc
- Chromium dc_layer_overlay.h exists; "List of overlays that are actually promoted": https://chromium.googlesource.com/chromium/src/+/HEAD/components/viz/service/display/dc_layer_overlay.h

- Chromium security FAQ: "The X11 protocol does not isolate clients from each other"; compromised GPU process attacking browser via X11 not a security bug: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/security/faq.md
- ScreenCast portal: Start() "typically presents a dialog to the user": https://flatpak.github.io/xdg-desktop-portal/docs/doc-org.freedesktop.portal.ScreenCast.html
- GNOME 49 (September 2025) disabled the X11 session; X11 apps via XWayland: https://www.gamingonlinux.com/2025/09/gnome-49-released-with-new-video-player-document-viewer-improved-app-store-and-disabled-x11-session
- GNOME 49 (September 2025) disabled the X11 session; X11 apps via XWayland: https://blogs.gnome.org/alatiera/2025/06/
- Wayland begun 2008 by Kristian Høgsberg; Wayland and Weston 1.0 announced 22 Oct 2012: https://lists.freedesktop.org/archives/wayland-devel/2012-October/005967.html
- Wayland begun 2008 by Kristian Høgsberg; Wayland and Weston 1.0 announced 22 Oct 2012: https://www.phoronix.com/news/MTIyMTE
- Mac OS X in stores 24 March 2001 with Quartz and Aqua: https://www.apple.com/newsroom/2001/03/21Mac-OS-X-Hits-Stores-This-Weekend/
- XWayland is a single Wayland client multiplexing all X11 clients, so X11 apps can reach each other but not native Wayland clients: https://lists.freedesktop.org/archives/wayland-devel/2014-January/012714.html
- iOS composition by Core Animation's render server, a separate process; render server in backboardd from iOS 6: https://www.wwdcnotes.com/notes/wwdc14/419
- iOS composition by Core Animation's render server, a separate process; render server in backboardd from iOS 6: https://nonstrict.eu/wwdcindex/wwdc2012/238/

Chapter 6. The Glass: Display, Color, HDR and Timing

- VESA added Adaptive-Sync to DisplayPort 1.2a, 12 May 2014; part of eDP since 2009: https://vesa.org/featured-articles/vesa-adds-adaptive-sync-to-popular-displayport-video-standard/
- ProMotion introduced with iPad Pro models, 5 June 2017, up to 120 Hz: https://www.macstories.net/news/apple-updates-ipad-pro-lineup-with-new-10-5-and-12-9-models/
- sRGB created 1996 by HP and Microsoft; IEC 61966-2-1 published 18 Oct 1999: https://webstore.iec.ch/publication/6169
- Windows: Advanced Color inactive = no color management, assumes sRGB; active = system color management; "If there is no profile, then EDID colorimetry data will be ...: https://learn.microsoft.com/en-us/windows/win32/wcs/advanced-color-icc-profiles
- Auto Color Management for HDR since Windows 10 1709; SDR in Windows 11 22H2 on qualifying hardware: https://devblogs.microsoft.com/directx/auto-color-management/

- Android wide color: colorMode="wideColorGamut", Android 8.0; not granted if display not color-managed; "more memory and GPU processing for screen composition": https://developer.android.com/training/wide-color-gamut
- wayland-protocols 1.41 (17 Feb 2025) added the color-management protocol; Mutter (GNOME 48) support: https://lists.freedesktop.org/archives/wayland-devel/2025-February/043980.html
- wayland-protocols 1.41 (17 Feb 2025) added the color-management protocol; Mutter (GNOME 48) support: https://www.phoronix.com/news/GNOME-wp_color_management_v1
- BT.2100-0 approved July 2016; PQ and HLG: https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.2100-0-201607-S!!PDF-F.pdf
- HDR content "isn't clamped unless it's composed to an SDR desktop": https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/for-best-performance--use-dxgi-flip-model
- UIScreen currentEDRHeadroom: headroom limits rendered content and changes with configuration: https://developer.apple.com/documentation/uikit/uiscreen/currentedrheadroom
- ColorSync Utility: change a device's color profile: https://support.apple.com/guide/colorsync-utility/toc
- Desktop Duplication API protects protected video; pointer delivered separately: https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/desktop-dup-api
- HDCP specification Revision 1.0, 17 Feb 2000, Intel copyright: https://digital-cp.com/sites/default/files/specifications/HDCP10.pdf
- Intel support: second display "goes out and does not function" during HDCP playback; fixed by newer driver: https://www.intel.com/content/www/us/en/support/articles/000099974/graphics.html
- RTINGS long-term test: long periods of static content cause burn-in on OLED TVs; LG screen shift and logo luminance adjustment: https://www.rtings.com/tv/learn/permanent-image-retention-burn-in-lcd-oled
- TDR: default timeout two seconds; "a screen flicker"; "Display driver stopped responding and has recovered"; legacy apps "might just render black"; app "must release and …: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/timeout-detection-and-recovery
- TDR: default timeout two seconds; "a screen flicker"; "Display driver stopped responding and has recovered"; legacy apps "might just render black"; app "must release and …: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/tdr-registry-keys

- Microsoft screen-capture guide: on HDR systems use DXGI_FORMAT_R16G16B16A16_FLOAT in the capture pipeline to avoid overclipping ("looks washed out"): https://learn.microsoft.com/windows/uwp/audio-video-camera/screen-capture
- "Windows 10, version 1709 first shipped Advanced Color support for HDR displays"; Advanced Color for "specially provisioned SDR displays with the Windows 11, version ...: https://learn.microsoft.com/en-us/windows/win32/direct3darticles/high-dynamic-range
- HDR capture: "pixel overclipping (i.e. the captured content looks washed out)", DXGI_FORMAT_R16G16B16A16_FLOAT: https://learn.microsoft.com/en-us/windows/apps/develop/media-authoring-processing/screen-capture

Chapter 7. Processes, Sandboxes and the GPU as the New Kernel

- Chrome launched 2 Sept 2008: https://developers.googleblog.com/2008/09/google-chrome-chromium-and-v8-launch.html
- Multi-process architecture post, 11 Sept 2008: browser, renderer, plug-in processes; renderer per site instance; later memory post on the trade-off: https://blog.chromium.org/2008/09/multi-process-architecture.html
- Windows sandbox: renderer restricted token, Untrusted integrity, "almost all resources that the renderer process uses have been acquired by the Browser"; job object per ...: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md
- Linux: layer-1 (namespaces), layer-2 seccomp-bpf to "restrict access from a process to the attack surface of the kernel": https://chromium.googlesource.com/chromium/src/+/2f7dfeb1/docs/linux/sandboxing.md
- macOS: Seatbelt / sandbox(7), (deny default), policies named per process type: https://chromium.googlesource.com/chromium/src/+/HEAD/sandbox/mac/README.md
- Android WebView multiprocess from Android 8.0; low-RAM fallback; onRenderProcessGone: https://developer.android.com/develop/ui/views/layout/webapps/manage-webview-memory
- Android WebView multiprocess from Android 8.0; low-RAM fallback; onRenderProcessGone: https://developer.android.com/guide/webapps/managing-webview
- Spectre disclosed 3 Jan 2018 (Project Zero): https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
- Site Isolation "enabled by default for all sites in Chrome 67"; "about 10-13%"; Chrome 77 Android login sites, 2 GB+, "about 3-5%"; "not yet supported in Android WebView": https://www.chromium.org/Home/chromium-security/site-isolation/

- Site Isolation "enabled by default for all sites in Chrome 67"; "about 10-13%"; Chrome 77 Android login sites, 2 GB+, "about 3-5%"; "not yet supported in Android WebView": https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html
- Firefox Fission released to desktop in Firefox 95 (7 Dec 2021): https://wiki.mozilla.org/Project_Fission
- WebView2 process group: "a single browser process", renderers, GPU and helpers; tied to one user data folder; several folders = several collections; ProcessFailed; ...: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/process-model
- GPU process introduced Aug 2010: "a new process (yes, another one) called the GPU process", commands to OpenGL or Direct3D via ANGLE: https://blog.chromium.org/2010/08/chromium-graphics-overhaul.html
- Command buffer: client "writes commands into some shared memory"; service validates each command; "Assume the client can go rogue"; "Graphics systems in OSes have gaping ...: https://www.chromium.org/developers/design-documents/gpu-command-buffer/
- WebKit Source/WebKit/GPUProcess directory exists (EntryPoint, media, webrtc, graphics): https://github.com/WebKit/WebKit/tree/main/Source/WebKit/GPUProcess
- Chromium security FAQ: X11 does not isolate clients; GPU-process-to-browser via X11 not a security bug: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/security/faq.md
- WebView2 flags: no-sandbox, disable-site-isolation-trials, disable-gpu; AdditionalBrowserArguments; WEBVIEW2_ADDITIONAL_BROWSER_ARGUMENTS env var and registry keys; ...: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/webview-features-flags
- Electron 20 (2022): renderers sandboxed by default unless nodeIntegration: true or sandbox: false: https://www.electronjs.org/blog/electron-20-0
- Electron 20 (2022): renderers sandboxed by default unless nodeIntegration: true or sandbox: false: https://www.electronjs.org/docs/latest/tutorial/sandbox
- Launch default was process-per-site-instance (new process for address-bar navigation to another site and for other instances in other tabs; link clicks and cross-site ...: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/process_model_and_site_isolation.md
- "Android WebView does not yet support multiple renderer processes or out-of-process iframes"; web site instance definition; process reuse, desktop "soft" process limit, ...: https://chromium.googlesource.com/chromium/src/+/f89ec29af8b53c5a1ac18a7946e643d61eff8562/docs/process_model_and_site_isolation.md

- WebView: "generally only one renderer process per browser context", "little process-level site isolation in WebView today": https://chromium.googlesource.com/chromium/src/+/f89ec29af8b53c5a1ac18a7946e643d61eff8562/android_webview/SECURITY.md
- Sandbox doc: "At M75 the browser … Medium … GPU process at Low … isolated renderers at Untrusted"; "will change over time": https://chromium.googlesource.com/chromium/src/+/f89ec29af8b53c5a1ac18a7946e643d61eff8562/docs/design/sandbox.md
- CEF sandbox setup: Windows requires linking cef_sandbox and passing cef_sandbox_info_create() to CefExecuteProcess/CefInitialize; "Starting with M138" use bootstrap.exe …: https://github.com/chromiumembedded/cef/blob/master/docs/sandbox_setup.md
- Qt WebEngine "provides out-of-the-box sandboxing support for Chromium render processes"; Linux needs seccomp-bpf and unprivileged user namespaces; …: https://doc.qt.io/qt-6/qtwebengine-platform-notes.html

Chapter 8. Input and Hit-Testing: Who Gets the Click

- "single system message queue and one thread-specific message queue for each GUI thread"; input posted to the queue of the thread that created the window; …: https://learn.microsoft.com/en-us/windows/win32/winmsg/about-messages-and-message-queues
- UIPI (Vista): lower-integrity process cannot SendMessage/PostMessage to higher; messages silently dropped: https://learn.microsoft.com/en-us/archive/blogs/vishalsi/what-is-user-interface-privilege-isolation-uipi-on-vista
- UAC "Switch to the secure desktop when prompting for elevation", enabled by default: https://learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/settings-and-configuration
- LLMHF_INJECTED / LLMHF_LOWER_IL_INJECTED flags in MSLLHOOKSTRUCT: https://learn.microsoft.com/en-us/windows/win32/api/winuser/ns-winuser-msllhookstruct
- DOM isTrusted: "dispatched by the user agent (as opposed to using dispatchEvent())": https://dom.spec.whatwg.org/#dom-event-istrusted
- Chromium viz hit-test aggregator and manager in components/viz/service/hit_test/; region flags for child surfaces (OOPIF): https://chromium.googlesource.com/chromium/src/+/89.0.4389.72/components/viz/service/hit_test/hit_test_aggregator.h
- Clickjacking coined by Jeremiah Grossman and Robert Hansen in 2008: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/09-Testing_for_Clickjacking
- IE8 X-Frame-Options (IEBlog, January 2009); RFC 7034 (October 2013): https://learn.microsoft.com/en-us/archive/blogs/ie/ie8-security-part-vii-clickjacking-defenses

- IE8 X-Frame-Options (IEBlog, January 2009); RFC 7034 (October 2013): https://www.rfc-editor.org/rfc/rfc7034
- Cloak and Dagger: Fratantonio, Qian, Chung, Lee; IEEE S&P, May 2017; overlay plus accessibility; 20-subject study, none noticed: https://www.eurekalert.org/news-releases/489174
- Cloak and Dagger: Fratantonio, Qian, Chung, Lee; IEEE S&P, May 2017; overlay plus accessibility; 20-subject study, none noticed: https://cloak-and-dagger.org/
- filterTouchesWhenObscured: framework discards touches when the window is obscured at the touched location: https://developer.android.com/reference/android/view/View
- Android 12 untrusted touch blocking, exceptions, 0.8 maximum obscuring opacity: https://developer.android.com/about/versions/12/behavior-changes-all#untrusted-touch-events
- HIDE_OVERLAY_WINDOWS added in Android 12; protection against cloak-and-dagger: https://developer.android.com/security/fraud-prevention/activities
- HIDE_OVERLAY_WINDOWS added in Android 12; protection against cloak-and-dagger: https://developer.android.com/about/versions/12/features
- Commit 2e7fdf11ec (Xianda Sun, 28 Nov 2018): its "about 1%" describes async hit tests caused by one defect (draw-quad hit-test data provider used when ...: https://chromium.googlesource.com/chromium/src/+/2e7fdf11ecb0fe2c64f9af9fd77c3c2124675b18
- IInvokeProvider::Invoke "Sends a request to activate a control and initiate its single, unambiguous action" (provider-side; no pointer input): https://learn.microsoft.com/en-us/windows/win32/api/uiautomationcore/nf-uiautomationcore-iinvokeprovider-invoke
- Invoke control pattern: "Invoking a control is generally performed by clicking or double-clicking or pressing ENTER..." (the pattern is the semantic equivalent, not the ...: https://learn.microsoft.com/en-us/windows/win32/winauto/uiauto-implementinginvoke
- WebAuthn Level 3 (W3C Recommendation, 25 August 2026): "User Verification: The technical process by which an authenticator locally authorizes ... through a touch plus ...: https://www.w3.org/TR/webauthn-3/

Chapter 9. Capture, Remote Display and the Trusted Pixel

- SetWindowDisplayAffinity: WDA_MONITOR and WDA_EXCLUDEFROMCAPTURE wording; WDA_EXCLUDEFROMCAPTURE from Windows 10 2004; "no guarantee ... for example where someone takes ...: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowdisplayaffinity
- Desktop Duplication replaces XDDM mirror drivers in Windows 8; dirty and move rects; coalescing; separate pointer shape must be drawn: https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/desktop-dup-api

- Windows.Graphics.Capture: system picker, yellow border; HDR "washed out" with 8-bit format: https://learn.microsoft.com/en-us/windows/apps/develop/media-authoring-processing/screen-capture
- GraphicsCaptureSession introduced Windows 10 1803 (10.0.17134); IsBorderRequired; IsCursorCaptureEnabled (2004): https://learn.microsoft.com/en-us/uwp/api/windows.graphics.capture.graphicscapturesession
- ScreenCaptureKit introduced macOS 12.3, WWDC22 session 10156: https://developer.apple.com/videos/play/wwdc2022/10156/
- sharingType none still captured by ScreenCaptureKit on macOS 15.4+ (developer report, July 2025); Apple DTS: "At this time there are no public APIs for preventing screen ...: https://developer.apple.com/forums/thread/792152
- UIScreen.isCaptured from iOS 11 (detection only): https://developer.apple.com/documentation/uikit/uiscreen/iscaptured
- xdg-desktop-portal ScreenCast: Start() "typically presents a dialog"; returns PipeWire streams: https://flatpak.github.io/xdg-desktop-portal/docs/doc-org.freedesktop.portal.ScreenCast.html
- MediaProjection added API level 21; consent per session (single-use token): https://developer.android.com/reference/android/media/projection/MediaProjection
- MediaProjection added API level 21; consent per session (single-use token): https://developer.android.com/media/grow/media-projection
- Android 14 QPR2 app screen sharing (single app): https://developer.android.com/about/versions/14/features/app-screen-sharing
- Android 14 screenshot detection; Android 15 screen recording detection: https://developer.android.com/about/versions/14/features/screenshot-detection
- Android 14 screenshot detection; Android 15 screen recording detection: https://developer.android.com/about/versions/15/features
- FLAG_SECURE (API 1) wording; screenshot "is blank"; "isn't reliable in preventing overlay attacks": https://developer.android.com/reference/android/view/WindowManager.LayoutParams#FLAG_SECURE
- FLAG_SECURE (API 1) wording; screenshot "is blank"; "isn't reliable in preventing overlay attacks": https://developer.android.com/security/fraud-prevention/activities
- Android Protected Confirmation: Android 9, API 28; "high confidence ... even if the Android framework (including the kernel) was compromised"; needs dedicated hardware; ...: https://developer.android.com/privacy-and-security/security-android-protected-confirmation

- Android Protected Confirmation: Android 9, API 28; "high confidence … even if the Android framework (including the kernel) was compromised"; needs dedicated hardware; …: https://developer.android.com/reference/android/security/ConfirmationPrompt
- PSD2 RTS Article 5(1)(a) to (d) and 5(2)(a)(b): https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32018R0389
- Azure Virtual Desktop default chroma 4:2:0: https://learn.microsoft.com/en-us/azure/virtual-desktop/configure-default-chroma-value
- iOS isSecureTextEntry layer workaround: developer forum question (Jul 2025) calls it a workaround because "there's no public API to block screenshots"; no Apple …: https://developer.apple.com/forums/thread/792624
- GraphicsCaptureSession.IsBorderRequired: Windows 10 version 2104, "introduced in 10.0.20348.0", UniversalApiContract v12; requires …: https://learn.microsoft.com/en-us/uwp/api/windows.graphics.capture.graphicscapturesession.isborderrequired
- MediaProjection guide: "Android 14 (API level 34) introduces app screen sharing"; Android 14 feature page and blog (11 Mar 2024): "On Android 14 QPR2, users can share or …: https://android-developers.googleblog.com/2024/03/enhanced-screen-sharing-capabilities-in-android-14.html
- IGraphicsCaptureItemInterop, "Desktop interop interface for the GraphicsCaptureItem activation factory"; CreateForWindow "Targets a single window", CreateForMonitor …: https://learn.microsoft.com/en-us/windows/win32/api/windows.graphics.capture.interop/nn-windows-graphics-capture-interop-igraphicscaptureiteminterop

Chapter 10. When the User Is an Agent

- Core-AAM 1.2 maps to MSAA with IAccessible2 1.3, UIA, ATK/AT-SPI, macOS AX API, Android accessibility API (Candidate Recommendation Draft, 5 Nov 2026): https://www.w3.org/TR/core-aam-1.2/
- MSAA released 1997 as add-on to Windows 95: https://en.wikipedia.org/wiki/Microsoft_Active_Accessibility
- UI Automation first released with Windows Vista; Windows 7 native providers and bridge to Active Accessibility clients: https://learn.microsoft.com/en-us/archive/blogs/tsfaware/the-history-of-ui-automation
- UIA description quotes; XP requires .NET Framework 3.0: https://learn.microsoft.com/en-us/windows/win32/winauto/entry-uiauto-win32
- AT-SPI moved from CORBA to D-Bus (announced 2008; Fedora 14 shipped it during the GNOME 3 transition): https://fedoraproject.org/wiki/Features/AtSpiTwo

- AT-SPI moved from CORBA to D-Bus (announced 2008; Fedora 14 shipped it during the GNOME 3 transition): https://lists.gnome.org/archives/gnome-accessibility-list/2008-November/msg00021.html
- AccessibilityService added API 4; window content as a tree of AccessibilityWindowInfo and AccessibilityNodeInfo; dispatchGesture; ERROR_TAKE_SCREENSHOT_SECURE_WINDOW: https://developer.android.com/reference/android/accessibilityservice/AccessibilityService
- setAccessibilityDataSensitive added API 34 (Android 14); limits to services with isAccessibilityTool: https://developer.android.com/sdk/api_diff/34/changes/android.view.View
- setAccessibilityDataSensitive added API 34 (Android 14); limits to services with isAccessibilityTool: https://developer.android.com/reference/android/view/View
- WCAG 2.1 SC 2.5.3 Label in Name (Level A) and intent quote: https://www.w3.org/WAI/WCAG21/Understanding/label-in-name.html
- aria-hidden removes content from the accessibility API: https://www.w3.org/TR/wai-aria-1.2/#aria-hidden
- Anthropic computer use public beta, 22 Oct 2024: https://www.anthropic.com/news/3-5-models-and-computer-use
- Computer-use docs: prompt injection warning quotes; classifiers scan screenshots: https://platform.claude.com/docs/en/agents-and-tools/tool-use/computer-use-tool
- OpenAI Operator, 23 Jan 2025; integrated into ChatGPT agent 17 July 2025: https://openai.com/index/introducing-operator/
- Greshake et al., indirect prompt injection, ACM AISec 2023: https://arxiv.org/abs/2302.12173
- Cloak and Dagger (overlay plus accessibility), IEEE S&P 2017: https://cloak-and-dagger.org/
- Display affinity page covers display and capture only; no mention of UI Automation: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowdisplayaffinity
- EN 301 549 references WCAG (EU accessibility procurement standard): https://www.etsi.org/deliver/etsi_en/301500_301599/301549/
- European Accessibility Act, Directive (EU) 2019/882, applies from 28 June 2025, covering banking and e-commerce services: https://eur-lex.europa.eu/eli/dir/2019/882/oj
- AccessibilityNodeInfo "Added in API level 14": https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo

- AccessibilityWindowInfo "Added in API level 21": https://developer.android.com/reference/android/view/accessibility/AccessibilityWindowInfo

Chapter 11. The Browser Inside the App: Eight Runtimes on One Ledger

- WebView2 evergreen vs fixed; "modified Microsoft Edge binaries"; evergreen preinstalled on Windows 11; hard links with Edge on eligible systems ("disk footprint, memory, ...: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/evergreen-vs-fixed-version
- Evergreen receives the same updates as Edge Stable and security notes; new version used "when your WebView2 app is restarted... if your app runs continuously, your app ...: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/distribution
- Process group = one browser process, renderers, GPU and helpers, tied to one user data folder; multiple folders = multiple groups: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/process-model
- Chrome 4-week milestones from Chrome 94; Extended Stable every 8 weeks for enterprises and embedders (post 4 Mar 2021): https://blog.chromium.org/2021/03/speeding-up-release-cycle.html
- Electron: major every 8 weeks; "latest three stable major versions are supported"; only latest minor supported. "About 24 weeks" is the author's arithmetic (3 x 8): https://www.electronjs.org/docs/latest/tutorial/electron-timelines
- CEF branch support: ~2 weeks stable, ~8 weeks extended, ~9 months LTC/LTS; legacy none; "likely be a delay": https://chromiumembedded.github.io/cef/branches_and_building
- CEF repository first commit 11f5769f19, 2008-11-26 "Initial directory structure"; libcef initial commit 2008-12-02 (GitHub API on chromiumembedded/cef commit history, ...: https://api.github.com/repos/chromiumembedded/cef/commits
- Qt WebEngine Chromium version policy, cherry-picked security patches, "next patch release is sped up"; Windows, Linux, macOS only: https://doc.qt.io/qt-6/qtwebengine-overview.html
- Apple Guideline 2.5.6 text: https://developer.apple.com/app-store/review/guidelines/
- EU: iOS 17.4; Web Browser Engine Entitlement (browser apps), Embedded Browser Engine Entitlement (in-app browsing, defined as content "that would be accessible and work ...: https://developer.apple.com/support/alternative-browser-engines/
- Japan: iOS 26.2; browser apps; embedded entitlement only for browser engine stewards: https://developer.apple.com/support/alternative-browser-engines-jp/
- Tauri: WebView2 / WKWebView "updated with the regular OS updates", unsupported macOS no WebKit updates / WebKitGTK "check your distro's repositories" / Android system ...: https://v2.tauri.app/reference/webview-versions/

- "Since Android Lollipop, WebView has been an updatable system component"; GMS vendors "must use Google's provided WebView configuration"; AOSP default …: https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/docs/webview-providers.md
- Monochrome/Trichrome "no longer recommended"; Monochrome not compatible with Q+ "due to changes made to support Trichrome": https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/docs/aosp-system-integration.md
- Chrome 116.0.5845.187/.188 on 11 Sep 2023; CVE-2023-4863 reported by Apple SEAR and Citizen Lab on 2023-09-06; "aware that an exploit … exists in the wild": https://chromereleases.googleblog.com/2023/09/stable-channel-update-for-desktop_11.html
- NVD CVE-2023-4863 published 2023-09-12; CISA KEV added 2023-09-13; affects Chrome < 116.0.5845.187 and libwebp < 1.3.2: https://nvd.nist.gov/vuln/detail/CVE-2023-4863
- CVE-2023-41064 fixed in iOS 16.6.1 etc.; NVD published 2023-09-07: https://nvd.nist.gov/vuln/detail/CVE-2023-41064
- Citizen Lab BLASTPASS report, 7 Sep 2023: https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/
- Edge 116.0.1938.81 fix for CVE-2023-4863, 12 Sep 2023: https://learn.microsoft.com/en-us/deployedge/microsoft-edge-relnotes-security
- Electron v26.2.1 published 2023-09-12T19:23Z; v25.8.1, v24.8.3, v22.3.24 published 2023-09-13 (UTC) with "Security: backported fix for CVE-2023-4863" (22/24/25 notes); …: https://api.github.com/repos/electron/electron/releases/tags/v22.3.24
- Electron v26.2.1 published 2023-09-12T19:23Z; v25.8.1, v24.8.3, v22.3.24 published 2023-09-13 (UTC) with "Security: backported fix for CVE-2023-4863" (22/24/25 notes); …: https://security.snyk.io/vuln/SNYK-JS-ELECTRON-5892810
- libwebp 1.3.2 on 9/13/2023, "security fix for lossless decoder (chromium: #1479274, CVE-2023-4863)": https://chromium.googlesource.com/webm/libwebp/+/refs/heads/main/NEWS
- Debian DSA-5497-1 libwebp, 13 Sep 2023: https://www.debian.org/security/2023/dsa-5497
- Qt advisory, 22 Sep 2023: WebP image plugin, fixed 5.15.16, 6.2.10, 6.5.3, 6.6.0: https://lists.qt-project.org/pipermail/announce/2023-September/000444.html
- Android Security Bulletin Oct 2023 (published 2 Oct 2023): CVE-2023-4863 "limited, targeted exploitation", patch level 2023-10-06 (System component, platform libwebp): https://source.android.com/docs/security/bulletin/2023-10-01
- Electron = Atom Shell; 1.0 on 11 May 2016; rename post "Atom Shell is now Electron" (2015): https://www.electronjs.org/blog/electron-1-0
- Microsoft to adopt Chromium, 6 Dec 2018 (Joe Belfiore): https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/

- New Edge available 15 Jan 2020: https://blogs.windows.com/windowsexperience/2020/01/15/new-year-new-browser-the-new-microsoft-edge-is-out-of-preview-and-now-available-for-download/
- WebView2 GA for Win32, 19 Oct 2020; .NET and fixed version GA 20 Nov 2020: https://blogs.windows.com/msedgedev/2020/10/19/edge-webview2-general-availability/
- WebView2 GA for Win32, 19 Oct 2020; .NET and fixed version GA 20 Nov 2020: https://blogs.windows.com/msedgedev/2020/11/20/announcing-general-availability-of-microsoft-edge-webview2-for-net-and-fixed-version-distribution-mode/
- NVD published 2023-09-12T15:15; cisaExploitAdd 2023-09-13: https://services.nvd.nist.gov/rest/json/cves/2.0?cveId=CVE-2023-4863
- Debian DSA-5497-1 dated 13 Sep 2023 (header Wed, 13 Sep 2023 21:07 UTC): https://lists.debian.org/debian-security-announce/2023/msg00189.html
- Qt advisory 22 Sep 2023 covers the qtimageformats WebP plugin and does not mention Qt WebEngine; Qt 6.5.3 release notes list no CVE-2023-4863 under qtwebengine: https://code.qt.io/cgit/qt/qtreleasenotes.git/tree/qt/6.5.3/release-note.md
- Trichrome first shipped for Android Q; Monochrome used on N, O, P: https://chromium.googlesource.com/chromium/src/+/125.0.6422.5/docs/android_native_libraries.md
- Chrome "Stable Channel Update for Desktop" published 2023-09-11T10:53-07:00 (17:53 UTC 11 Sep): https://chromereleases.googleblog.com/feeds/posts/default?alt=json
- libwebp v1.3.2: release commit ca332209cb 2023-09-13 15:11 -0700 (22:11 UTC); tag 2023-09-13 18:49 -0700 (01:49 UTC 14 Sep); NEWS "9/13/2023". Command: git fetch …: https://chromium.googlesource.com/webm/libwebp
- webview-providers.md at main f89ec29af8 (committed 2026-11-18 04:47): ">= Q … Standalone AOSP WebView (default, preinstalled)" / "Standalone WebView Google Stable …: https://chromium.googlesource.com/chromium/src/+/f89ec29af8b53c5a1ac18a7946e643d61eff8562/android_webview/docs/webview-providers.md
- Android Security Bulletin October 2023, "Published October 2, 2023": CVE-2023-4863 (A-299477569, RCE, Critical) is listed under "2023-10-06 security patch level …: https://web.archive.org/web/20231010095514/https://source.android.com/docs/security/bulletin/2023-10-01
- Chrome for Android 117.0.5938.60, 12 Sep 2023: "Android releases contain the same security fixes as their corresponding Desktop release (Windows: 117.0.5938.62/.63…)"; …: https://chromereleases.googleblog.com/2023/09/
- Chrome for Android 117.0.5938.60, 12 Sep 2023: "Android releases contain the same security fixes as their corresponding Desktop release (Windows: 117.0.5938.62/.63…)"; …: https://chromereleases.googleblog.com/search/label/Chrome%20for%20Android

- Debian bookworm libwebkit2gtk-4.1-0 depends on libwebp7 (the system libwebp DSA-5497-1 updated): https://packages.debian.org/bookworm/libwebkit2gtk-4.1-0

Chapter 12. The Native Path: Composition, Skia and the Rust Renderers

- Composition visuals are "the visual tree structure that all other features of the Microsoft.UI.Composition API use and build on"; WinUI apps retrieve compositor from ...: https://learn.microsoft.com/en-us/windows/apps/develop/composition/composition-visual-tree
- DirectComposition "was introduced in Windows 8"; operates "independently of the UI thread": https://learn.microsoft.com/en-us/windows/win32/directcomp/directcomposition-portal
- Core Animation "first introduced in OS X v10.5": https://developer.apple.com/library/mac/documentation/graphicsimaging/Conceptual/Animation_Overview/MacOSXAnimTech/MacOSXAnimTech.html
- GTK 4.0 released December 2020; GSK scene graph: https://blog.gtk.org/2020/12/
- GTK 4.0 released December 2020; GSK scene graph: https://docs.gtk.org/gsk4/
- Qt Quick scene graph "traversed and rendered via a graphics API"; "dedicated render thread": https://doc.qt.io/qt-6/qtquick-visualcanvas-scenegraph.html
- Jetpack Compose 1.0, 28 July 2021: https://android-developers.googleblog.com/2021/07/jetpack-compose-announcement.html
- Skia "sponsored and managed by Google"; engine for Chrome, ChromeOS, Android, Flutter: https://skia.org/about/
- Graphite announced 8 July 2025, launch in Chrome on Apple Silicon Macs; Metal, Vulkan, D3D12: https://blog.google/chromium/introducing-skia-graphite-chromes/
- Impeller default on iOS in Flutter 3.10 (May 2023); default on Android API 29+ in Flutter 3.27 (Dec 2024), OpenGL fallback: https://flutter.dev/blog/whats-new-in-flutter-3-10
- Impeller default on iOS in Flutter 3.10 (May 2023); default on Android API 29+ in Flutter 3.27 (Dec 2024), OpenGL fallback: https://flutter.dev/blog/whats-new-in-flutter-3-27
- Impeller default on iOS in Flutter 3.10 (May 2023); default on Android API 29+ in Flutter 3.27 (Dec 2024), OpenGL fallback: https://docs.flutter.dev/perf/impeller
- wgpu: "cross-platform, safe, pure-Rust graphics API"; Vulkan, Metal, D3D12, OpenGL; "core of the WebGPU integration in Firefox, Servo, and Deno": https://github.com/gfx-rs/wgpu
- Vello: "GPU compute-centric 2D renderer"; compute renderer "an experimental implementation"; Vello CPU "currently overall more mature": https://github.com/linebender/vello

- Compose semantics: "when you're adding custom low-level composables, you have to manually provide its semantics"; calendar example ("Calendar"): https://developer.android.com/develop/ui/compose/accessibility/semantics
- Flutter Semantics widget and SemanticsNode / semantics tree: https://api.flutter.dev/flutter/widgets/Semantics-class.html
- AccessKit purpose quote; UIA, NSAccessibility, UIAccessibility, AT-SPI, Android: https://github.com/AccessKit/accesskit
- Servo "Created by Mozilla Research in 2012", "first major Rust codebase other than the compiler itself"; stewardship to Linux Foundation 2020: https://www.igalia.com/2023/09/07/The-Servo-project-is-joining-Linux-Foundation-Europe.html
- Servo "Created by Mozilla Research in 2012", "first major Rust codebase other than the compiler itself"; stewardship to Linux Foundation 2020: https://servo.org/about/
- Servo joins Linux Foundation Europe, 7 Sep 2023; renewed activity led by Igalia: https://linuxfoundation.eu/newsroom/servo-web-rendering-engine-joins-linux-foundation-europe
- Linux Foundation to host Servo, 17 Nov 2020; Stylo/Quantum 2017 and WebRender credited: https://hacks.mozilla.org/2020/11/foundations-for-the-future/
- Linux Foundation to host Servo, 17 Nov 2020; Stylo/Quantum 2017 and WebRender credited: https://linuxfoundation.org/press-release/open-source-web-engine-servo-to-be-hosted-at-linux-foundation
- Mozilla restructuring, about 250 roles, 11 Aug 2020: https://blog.mozilla.org/en/mozilla/changing-world-changing-mozilla/
- Flutter 3.47 (12 Aug 2026): "Metal on macOS and Vulkan on Windows and Linux": https://flutter.dev/blog/whats-new-in-flutter-3-47
- iOS: Core Animation render server is the backboardd process: https://developer.apple.com/documentation/visionos/understanding-the-visionos-render-pipeline
- Servo v0.0.1 tag commit 721214fbe4 dated 2025-10-19, GitHub release published 2025-10-20; v0.8.0 tag commit 4e136d8ef3 dated 2026-11-05, release published 2026-11-09: https://api.github.com/repos/servo/servo/releases
- Flutter 3.29 (12 Feb 2025): devices without functional Vulkan "will fall back to Impeller running on OpenGLES instead of using Skia"; Skia removed from iOS backend: https://flutter.dev/blog/whats-new-in-flutter-3-29
- Deno 1.8 (2 Mar 2021) shipped WebGPU built on wgpu: https://deno.com/blog/v1.8
- Firefox 141 (22 Jul 2025): "Enabled the WebGPU API … on Windows": https://www.firefox.com/en-US/firefox/141.0/releasenotes/

- AppKit 10.14: windows of apps linked on the 10.14 SDK "are displayed using Core Animation"; "This doesn't mean that all views are layer-backed": https://developer.apple.com/documentation/macos-release-notes/appkit-release-notes-for-macos-10_14
- Windows App SDK (WinUI 3) is not serviced with the OS: framework-dependent apps get "Servicing updates … via the Windows App SDK Framework package"; self-contained …: https://learn.microsoft.com/en-us/windows/apps/package-and-deploy/deploy-overview

Chapter 13. Who Pays for the Engine: The Money Story

- "Google is a monopolist, and it has acted as one to maintain its monopoly. It has violated Section 2 of the Sherman Act.": https://storage.courtlistener.com/recap/gov.uscourts.dcd.223205/gov.uscourts.dcd.223205.1033.0_2.pdf
- Remedies top-line determinations (exclusive contracts barred incl. Search, Chrome, Assistant, Gemini app; one-year term; no Chrome divestiture; payments not barred, …: https://storage.courtlistener.com/recap/gov.uscourts.dcd.223205/gov.uscourts.dcd.223205.1436.0.pdf
- Mozilla 2024: royalties $498,218K; total revenue and support $680,409K (2023: $494,874K / $653,012K); software development $290,448K; "Approximately 86% and 85% of …: https://stateof.mozilla.org/pdf/Mozilla%20Fdn%202024%20-%20AuditedFinancials.pdf
- Microsoft 6 Dec 2018 Chromium announcement and quoted reasons: https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/
- New Edge 15 Jan 2020: https://blogs.windows.com/windowsexperience/2020/01/15/new-year-new-browser-the-new-microsoft-edge-is-out-of-preview-and-now-available-for-download/
- Guideline 2.5.6; EU entitlements (iOS 17.4, browser and in-app browsing, in-app browsing definition, EU-only distribution); Japan (iOS 26.2, steward-only embedded): https://developer.apple.com/app-store/review/guidelines/
- Guideline 2.5.6; EU entitlements (iOS 17.4, browser and in-app browsing, in-app browsing definition, EU-only distribution); Japan (iOS 26.2, steward-only embedded): https://developer.apple.com/support/alternative-browser-engines/
- Guideline 2.5.6; EU entitlements (iOS 17.4, browser and in-app browsing, in-app browsing definition, EU-only distribution); Japan (iOS 26.2, steward-only embedded): https://developer.apple.com/support/alternative-browser-engines-jp/
- Skia "sponsored and managed by Google": https://skia.org/about/
- Electron in OpenJS incubation Dec 2019; Impact Project 23 Jun 2020: https://www.electronjs.org/blog/electron-joins-openjsf
- Electron in OpenJS incubation Dec 2019; Impact Project 23 Jun 2020: https://www.electronjs.org/blog/electron-openjs-impact-project

- Edge Legacy (EdgeHTML) support ended 9 March 2021: https://learn.microsoft.com/en-us/lifecycle/products/microsoft-edge-legacy
- Igalia maintains the WebKitGTK and WPE ports: https://www.igalia.com/2023/10/30/WebKit-Contributors-Meeting-2023.html
- Igalia maintains the WebKitGTK and WPE ports: https://igalia.com/technology/browsers
- chromium/src head e52d393f6d (2026-11-17): https://chromium.googlesource.com/chromium/src.git
- WebKit head f31b755f63d (2026-11-17): https://github.com/WebKit/WebKit.git
- firefox head 524e4e57a3 (2026-11-18): https://github.com/mozilla-firefox/firefox.git

Chapter 14. The Dark Pages: Antitrust, Exploits and Tracking

- US v. Microsoft: filed 18 May 1998; findings of fact 5 Nov 1999; final judgment approved 12 Nov 2002: https://www.justice.gov/atr/case/us-v-microsoft-corporation-browser-and-middleware
- Findings of fact paras 161, 164, 174 (quotes): https://www.justice.gov/atr/us-v-microsoft-courts-findings-fact
- Conclusions of law 3 Apr 2000; final judgment (breakup) 7 Jun 2000: https://webarchive.library.unt.edu/web/20090831010535/http://justice.gov/atr/cases/ms_conclusions.htm
- D.C. Cir. en banc 28 Jun 2001, 253 F.3d 34: affirmed in part monopoly maintenance; "commingling of browser and operating system code" exclusionary under s.2; reversed ...: https://www.law.berkeley.edu/wp-content/uploads/2015/04/US-v-Microsoft-2001-excerpt.pdf
- Settlement Nov 2001 (stipulation 2 Nov 2001): https://www.justice.gov/atr/us-v-microsoft-corporation-settlement-information
- US v. Google memorandum opinion 5 Aug 2024 (ECF 1033): "In 2021, Google paid out a total of $26.3 billion in revenue share": https://www.cs.princeton.edu/courses/archive/fall24/cos109/dojvsgoogle.pdf
- CVE-2011-3402 NVD wording (win32k.sys TrueType, Word document or web page, Duqu Nov 2011): https://nvd.nist.gov/vuln/detail/CVE-2011-3402
- Advisory 2639658, 3 Nov 2011: https://learn.microsoft.com/en-us/security-updates/securityadvisories/2011/2639658
- MS11-087, 13 Dec 2011: https://learn.microsoft.com/en-us/security-updates/securitybulletins/2011/ms11-087
- "With Windows 10, GDI font parsing is no longer performed in kernel mode... fontdrvhost.exe... AppContainer... no Capabilities": https://techcommunity.microsoft.com/blog/microsoft-security-baselines/dropping-the-untrusted-font-blocking-setting/701068

- Black Hat USA 2016 (Weston): font parsing to user-mode sandbox, Anniversary Update: https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf
- Stagefright: Zimperium 27 Jul 2015, MMS, ~950 million devices, CVE-2015-1538 et al.: https://zimperium.com/blog/experts-found-a-unicorn-in-the-heart-of-android
- First Nexus Security Bulletin 13 Aug 2015, "most notably [via] MMS and browser playback of media": https://source.android.com/docs/security/bulletin/2015-08-01
- FORCEDENTRY: Citizen Lab 13 Sep 2021; Saudi activist; NSO Pegasus; fake .gif = PSD/PDF with JBIG2: https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/
- CVE-2021-30860 CoreGraphics integer overflow, iOS 14.8: https://nvd.nist.gov/vuln/detail/CVE-2021-30860
- Project Zero 15 Dec 2021, Beer and Gross; ".gif" quote; JBIG2 emulated computer, 70,000+ segment commands: https://projectzero.google/2021/12/a-deep-dive-into-nso-zero-click.html
- BlastDoor, Project Zero 28 Jan 2021: https://projectzero.google/2021/01/a-look-at-imessage-in-ios-14.html
- Lockdown Mode announced 6 Jul 2022; quote: https://www.apple.com/newsroom/2022/07/apple-expands-commitment-to-protect-users-from-mercenary-spyware/
- BLASTPASS: Citizen Lab 7 Sep 2023; PassKit attachments with malicious images; NSO Pegasus; DC civil society employee: https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/
- iOS 16.6.1: CVE-2023-41064 ImageIO buffer overflow, CVE-2023-41061 Wallet; KEV 11 Sep 2023: https://support.apple.com/en-us/HT213905
- Chrome 116.0.5845.187, 11 Sep 2023; CVE-2023-4863 reported by Apple SEAR and Citizen Lab 6 Sep; exploit in the wild: https://chromereleases.googleblog.com/2023/09/stable-channel-update-for-desktop_11.html
- NVD description change 27 Sep 2023 to libwebp; CVE-2023-5129 rejected as duplicate; KEV 13 Sep 2023: https://nvd.nist.gov/vuln/detail/CVE-2023-4863
- Mowery and Shacham, W2SP 2012: https://www.ieee-security.org/TC/W2SP/2012/papers/w2sp12-final4.pdf
- Acar et al. CCS 2014: "more than 5.5% of the crawled sites" (top 100,000): https://securehomes.esat.kuleuven.be/~gacar/persistent/index.html
- Fifield and Egelman, FC 2015: https://rd.springer.com/chapter/10.1007/978-3-662-47854-7_7

- DrawnApart, NDSS 2022: "up to 67% to the median tracking duration", "over 2,500 crowd-sourced devices": https://arxiv.org/abs/2201.09956
- Stone, Black Hat USA 2013, white paper July 2013: https://infocondb.org/con/black-hat/black-hat-usa-2013/pixel-perfect-timing-attacks-with-html5
- Kotcher, Pei, Jumde, Jackson, CCS 2013, DOI 10.1145/2508859.2516712: https://doi.org/10.1145/2508859.2516712
- GPU.zip, IEEE S&P 2024; Chrome pixel leak; iframes + SVG filters; Firefox/Safari not affected; vendors told March 2023; public late Sept 2023: https://www.hertzbleed.com/gpu.zip/
- FOF 161, 164, 174 quotes; D.C. Cir. en banc 28 Jun 2001, 253 F.3d 34, "commingling of browser and operating system code": https://www.justice.gov/atr/cases/f3800/msjudgex.htm
- Duqu: Symantec alerted 14 Oct 2011 and reported in October 2011; the Word/TrueType zero-day installer shown in November 2011 (Advisory 2639658, 3 Nov 2011): https://usenix.org/system/files/conference/leet12/leet12-final11.pdf

Chapter 15. Cost and Debt: The Patch Bill, the Memory Bill, the Legacy Bill

- Chromium 4-week majors from Chrome 94 (Q3 2021), Extended Stable every 8 weeks; post 4 Mar 2021: https://blog.chromium.org/2021/03/speeding-up-release-cycle.html
- Electron: "The latest three stable major versions are supported"; 8-week cadence: https://www.electronjs.org/docs/latest/tutorial/electron-timelines
- Electron 8-week cadence from Sept 2021 (Electron 15, 21 Sept 2021), on Chromium Extended Stable; post 14 Jul 2021: https://www.electronjs.org/blog/8-week-cadence
- WebView2: continuously running app keeps previous runtime until it releases environment objects or restarts; NewBrowserVersionAvailable: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/distribution
- WebView2: "a collection of WebView2 Runtime processes will be created for each of these user data folders": https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/process-model
- WebView2: Edge and runtime binaries hard-linked on eligible systems when same version: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/evergreen-vs-fixed-version
- Site Isolation: "about a 10-13% total memory overhead in real workloads"; Chrome 67 on Windows, Mac, Linux, Chrome OS, 99% of users; Google Security Blog 11 Jul 2018: https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html

- IE mode supported "through at least 2029", one year notice; IE11 out of support for certain OSes from 15 Jun 2022: https://learn.microsoft.com/en-us/lifecycle/faq/internet-explorer-microsoft-edge
- UIWebView: new apps rejected from April 2020 (notice 23 Dec 2019); update deadline extended beyond end of 2020 (8 Oct 2020): https://developer.apple.com/news/?id=12232019b
- UIWebView: new apps rejected from April 2020 (notice 23 Dec 2019); update deadline extended beyond end of 2020 (8 Oct 2020): https://developer.apple.com/news/?id=edwud51q
- RDP 8.1 AVC/H.264 mixed mode; RDP 10 AVC 444 with Windows 10 1511 (Nov 2015): https://techcommunity.microsoft.com/t5/microsoft-security-and/remote-desktop-protocol-rdp-10-avc-h-264-improvements-in-windows/ba-p/249588
- Policy names "Prioritize H.264/AVC 444 Graphics mode..." and "Configure H.264/AVC hardware encoding...": https://learn.microsoft.com/en-us/azure/virtual-desktop/graphics-enable-gpu-acceleration
- RemoteFX vGPU disabled July 2020 updates, removed April 2021 (KB4570006): https://support.microsoft.com/en-us/topic/kb4570006-update-to-disable-and-remove-the-remotefx-vgpu-component-in-windows-bbdf1531-7188-2bf4-0de6-641de79f09d2
- Citrix: "Thinwire is the default display remoting technology"; H.264/H.265: https://docs.citrix.com/en-us/citrix-virtual-apps-desktops/2203-ltsr/graphics/thinwire
- Citrix: hardware encoding default on; "falls back to CPU-based encoding using the software video codec": https://docs.citrix.com/en-us/citrix-virtual-apps-desktops/policies/reference/ica-policy-settings/graphics-policy-settings.html
- GNOME 49: dedicated X11 session disabled by default; X11 apps "continue to be supported via xwayland": https://release.gnome.org/49/developers/
- GNOME 50 dev releases: mutter 50.alpha "Drop the X11 backend": https://gitlab.gnome.org/GNOME/mutter/-/raw/main/NEWS
- GDI "can be used in all Windows-based applications": https://learn.microsoft.com/en-us/windows/win32/gdi/windows-gdi
- Direct2D "should be considered, as it takes better advantage of modern GPUs": https://learn.microsoft.com/en-us/windows/win32/direct2d/comparing-direct2d-and-gdi
- AOSP WebView prebuilt "is not currently updated on a regular schedule, and may have known security issues": https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/docs/aosp-system-integration.md

- Legacy WebKit WebView deprecated since macOS 10.14 ("No longer supported; please adopt WKWebView."): https://opensource.apple.com/source/WebKit/WebKit-7607.3.9/mac/WebView/WebView.h.auto.html
- Mutter NEWS, 50.alpha: "Drop the X11 backend [Bilal; !4505]"; GNOME 50 release notes dated 18 March 2026: https://release.gnome.org/50/

Chapter 16. The Screen Ahead: The OS, AI and the Renderer

- Google Security blog "Supporting the Use of Rust in the Chromium Project", 12 Jan 2023: https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html
- Fontations/Skrifa: Chrome 128 trial (color, CFF2); Chrome 133 all web fonts on Linux, Android, ChromeOS, fallback on Windows and Mac; post 19 Mar 2025: https://developer.chrome.com/blog/memory-safety-fonts
- Android memory-safety share 76% (2019) to 24% (2024), post 25 Sep 2024: https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html
- Android memory-safety below 20% for first time (2025), post 13 Nov 2025: https://blog.google/security/rust-in-android-move-fast-fix-things/
- Stylo default in Firefox 57 (14 Nov 2017): https://developer.mozilla.org/Firefox/Releases/57
- WebRender rollout begins Firefox 67 (21 May 2019), Windows 10 + NVIDIA subset: https://www.mozilla.org/firefox/67.0/releasenotes
- Servo joins Linux Foundation Europe, 7 Sep 2023: https://www.igalia.com/2023/09/07/The-Servo-project-is-joining-Linux-Foundation-Europe.html
- Servo v0.0.1, 20 Oct 2025: https://servo.org/blog/2025/10/20/servo-0.0.1-release/
- servo crate v0.1.0 embeddable library, 13 Apr 2026: https://servo.org/blog/2026/04/13/servo-0.1.0-release/
- Vello README: compute renderer "experimental"; Vello CPU "currently overall more mature"; pre-1.0: https://github.com/linebender/vello
- C2PA spec 2.3 dated 2026-01-05: https://spec.c2pa.org/specifications/specifications/2.3/specs/_attachments/C2PA_Specification.pdf
- Pixel 10 "first to have Content Credentials built in across every photo created by Pixel Camera", 10 Sep 2025: https://security.googleblog.com/2025/09/pixel-android-trusted-images-c2pa-content-credentials.html
- EU AI Act Art. 50 applies from 2 Aug 2026; Reg. (EU) 2026/1744 Art. 111(4): systems on market before 2 Aug 2026 comply with Art. 50(2) by 2 Dec 2026: https://eur-lex.europa.eu/eli/reg/2026/1744/oj
- Copilot+ PCs announced 20 May 2024, "40+ TOPS": https://blogs.microsoft.com/blog/2024/05/20/introducing-copilot-pcs/

- Copilot+ 16 GB minimum memory: https://support.microsoft.com/en-us/topic/windows-next-generation-ai-device-hardware-requirements-35782169-6eab-4d63-a5c5-c498c3037364
- Apple Foundation Models framework, WWDC 9 Jun 2025; ~3B on-device model (Apple tech report): https://arxiv.org/abs/2507.13575
- Gemini Nano runs in Android's AICore system service: https://developer.android.com/ai/gemini-nano
- Chrome Prompt API: 22 GB free disk; GPU "strictly more than 4 GB of VRAM" or 16 GB RAM and 4 cores: https://developer.chrome.com/docs/ai/prompt-api
- Generative UI, Google Research, 18 Nov 2025, Gemini 3 Pro: https://research.google/blog/generative-ui-a-rich-custom-visual-interactive-user-experience-for-any-prompt/
- MCP Apps proposal 21 Nov 2025 (Anthropic, OpenAI, MCP-UI); official extension 26 Jan 2026: https://blog.modelcontextprotocol.io/posts/2025-11-21-mcp-apps/
- MCP Apps proposal 21 Nov 2025 (Anthropic, OpenAI, MCP-UI); official extension 26 Jan 2026: https://blog.modelcontextprotocol.io/posts/2026-01-26-mcp-apps/
- OpenAI apps in ChatGPT and Apps SDK preview (built on MCP), 6 Oct 2025: https://openai.com/index/introducing-apps-in-chatgpt/
- WebGPU: Chrome 113, Firefox 141 on Windows, Safari 26: https://web.dev/blog/webgpu-supported-major-browsers
- Apple GPUs: "unified memory model in which the CPU and the GPU share system memory": https://developer.apple.com/documentation/metal/choosing-a-resource-storage-mode-for-apple-gpus
- WebGPU by default in Chrome 113 on ChromeOS, macOS, Windows: https://developer.chrome.com/blog/webgpu-release
- WebGPU enabled in Firefox 141 on Windows; first offered 22 Jul 2025: https://www.firefox.com/en-US/firefox/141.0/releasenotes/
- Safari 26.0: WebGPU "now shipping … for macOS, iOS, iPadOS, and visionOS"; before that only in Safari Technology Preview, so not default in Safari 18: https://webkit.org/blog/17333/webkit-features-in-safari-26-0/
- Safari 26.0: WebGPU "now shipping … for macOS, iOS, iPadOS, and visionOS"; before that only in Safari Technology Preview, so not default in Safari 18: https://webkit.org/blog/16993/
- Foundation Models framework announced 9 June 2025 (WWDC25): https://www.apple.com/newsroom/2025/06/apple-supercharges-its-tools-and-technologies-for-developers/

- Reg. (EU) 2026/1744 of 8 July 2026, OJ 24.7.2026, in force on the third day after publication (27 July 2026); amended Art. 111: comply with Art. 50(2) by 2 December …: https://eur-lex.europa.eu/eli/reg/2026/1744/oj/eng
- C2PA 2.3: PDF title page "2.3, 2026-01-05"; HTML version history "2.3 - December 2025": https://spec.c2pa.org/specifications/specifications/2.3/specs/C2PA_Specification.html
- MCP Apps official-extension post dated 26 Jan 2026: https://blog.modelcontextprotocol.io/tags/apps/
- Servo v0.1.0: tag commit 8032867a0d 2026-04-11, GitHub release and crates.io 2026-04-13; v0.8.0 tag 2026-11-05, crate 2026-11-07, release 2026-11-09: https://crates.io/api/v1/crates/servo/versions
- MSAA first released as a Windows 95 add-on in 1997, built in from Windows 98: https://learn.microsoft.com/en-us/windows/win32/winauto/microsoft-active-accessibility
- Pixel 10 launch post, 20 Aug 2025: "Our latest phones will be the first to have industry-standard C2PA Content Credentials built into the native camera app": https://blog.google/products-and-platforms/devices/pixel/pixel-10-camera-features/
- Anthropic open-sourced MCP on 25 Nov 2024 ("Today, we're open-sourcing the Model Context Protocol (MCP)"): https://www.anthropic.com/news/model-context-protocol

Appendix C. The Five-OS Reference Table

- WDA_EXCLUDEFROMCAPTURE introduced in Windows 10 version 2004; "unlike a security feature or an implementation of Digital Rights Management (DRM), there is no …: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowdisplayaffinity
- "As of Windows 8, DWM composition is always enabled.": https://learn.microsoft.com/en-us/windows/win32/api/dwmapi/nf-dwmapi-dwmiscompositionenabled
- Desktop composition introduced in Windows Vista: https://learn.microsoft.com/en-us/windows/win32/dwm/dwm-overview
- Windows.Graphics.Capture: Windows 10, version 1803 (10.0.17134.0): https://learn.microsoft.com/en-us/uwp/api/windows.graphics.capture.graphicscapturesession
- Desktop Duplication API: https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/desktop-dup-api
- DXGI flip model: https://learn.microsoft.com/en-us/windows/win32/direct3ddxgi/dxgi-flip-model
- ScreenCaptureKit introduced in macOS 12.3 (WWDC23 session 10136; SCWindow availability macOS 12.3): https://developer.apple.com/wwdc23/10136
- ScreenCaptureKit introduced in macOS 12.3 (WWDC23 session 10136; SCWindow availability macOS 12.3): https://developer.apple.com/documentation/screencapturekit

- UIScreen isCaptured: iOS 11.0; doc availability shows 11.0 to 27.0 (deprecated); FairPlay Streaming video blacked out in capture, audio included: https://developer.apple.com/documentation/uikit/uiscreen/iscaptured
- FLAG_SECURE API level 1: "treat the content of the window as secure, preventing it from appearing in screenshots or from being viewed on non-secure displays": https://developer.android.com/reference/android/view/WindowManager.LayoutParams#FLAG_SECURE
- MediaProjection added in API level 21: https://developer.android.com/reference/android/media/projection/MediaProjection
- SurfaceFlinger composes visible surfaces, wakes on VSync, asks HWC how to compose; ASurfaceControl added in Android 10: https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager
- SurfaceFlinger composes visible surfaces, wakes on VSync, asks HWC how to compose; ASurfaceControl added in Android 10: https://source.android.com/docs/core/graphics
- Android Protected Confirmation: Android 9 (API 28) on supported devices; signs the prompt text the user confirmed; "doesn't provide a secure information channel"; ...: https://developer.android.com/privacy-and-security/security-android-protected-confirmation
- Minikin uses HarfBuzz and ICU (Android text layout); hwui MinikinUtils glue: https://android.googlesource.com/platform/frameworks/minikin/
- xdg-desktop-portal ScreenCast: streams as PipeWire nodes; OpenPipeWireRemote: https://flatpak.github.io/xdg-desktop-portal/docs/doc-org.freedesktop.portal.ScreenCast.html
- GNOME 49 (Sept 2025): X11 session disabled by default (not deleted); removal planned for GNOME 50: https://release.gnome.org/49/developers/
- Chromium RenderingNG pipeline stages and processes (render, browser, Viz; compositor thread): https://developer.chrome.com/docs/chromium/renderingng-architecture
- Chromium shaping with HarfBuzz on all platforms (see Chapter 3 sources): https://harfbuzz.github.io/
- ScreenCaptureKit introduced at WWDC22 ("Meet ScreenCaptureKit", session 10156, "a brand-new framework on macOS"); availability macOS 12.3: https://developer.apple.com/videos/play/wwdc2022/10156/
- Japan, iOS 26.2: Web Browser Engine Entitlement for browser apps; Embedded Browser Engine Entitlement only for browser engine stewards: https://developer.apple.com/support/alternative-browser-engines-jp/
- iOS render server is backboardd: https://developer.apple.com/documentation/visionos/understanding-the-visionos-render-pipeline

- Hop 4 Windows: Fontations as web font fallback (Chrome 133), FreeType removed from Blink (Chrome 145): `https://developer.chrome.com/blog/memory-safety-fonts`
- GNOME 50 (18 Mar 2026), Mutter 50.alpha "Drop the X11 backend": `https://gitlab.gnome.org/GNOME/mutter/-/raw/main/NEWS`

THE SERIES

Every Pixel Matters

One book for the whole stack, then one book for each layer that decides what your customers see. The companion series, GDCD in Code, covers how to govern the layers you choose to own.

Book 1	**From Bit to Pixel**: the whole stack, history, own or rent
Book 2	**From Code to Glass**: the rendering layer, from code to glass (this book)
Book 3	**From Silicon to Screen**: security from the chip to the screen
Book 4	**Data in Motion**: storage, networking, distributed systems
Book 5	**The Owned Stack**: toolchain, IDE, building an owned system
Book 6	**The Pixel Runtime**: building our own runtime (planned)

Juan Cardena

Enterprise AI & Agentic Systems Architect · blog.jcardena.com

Download the latest edition at blog.jcardena.com/en/ebooks/from-code-to-glass.

Every Pixel Matters · Book 2 · From code to glass.