EVERY PIXEL MATTERS

BOOK 1



FROM THE BIT TO THE PIXEL

From Bit to Pixel

What to Own, What to Rent

Every screen your customers see is built by many owners, from the silicon to the pixel. This book walks the stack of Windows, Linux and macOS one layer at a time, tells how it got this way, and gives a CTO a plain way to decide which layers to own and which to rent, for the business outcome that matters.

Juan Cardena

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

CONTENTS

BEFORE YOU START

Reader's Brief: Security, Cost, Value

PROLOGUE

One Key Press, and One Question

PART I · THE STACK, LAYER BY LAYER

01 Silicon and Boot

02 The Kernel

03 Drivers, Storage and Network

04 Security, Identity and the Trusted Pixel

05 Runtimes and Languages

06 Graphics, Text and the Pixel

07 The Browser, the Second OS

08 Toolchain, Distribution and the Edges of an OS

PART II · HOW WE GOT HERE

09 Seventy Years of Operating Systems

10 Why the Winners Won

PART III · OWNING FROM BIT TO PIXEL

11 The Seams, in Numbers

12 The Price of Owning

13 AI Moves the Seams

14 Controlling the Bit and the Toolchain

15 What to Own, What to Rent

CLOSING

Epilogue: The Last Layer Is the Human

APPENDICES

A Glossary

B Timeline Poster

C Full Comparison Table

Reader's Brief: Security, Cost, 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, the questions each one brings, and the one tool the book builds layer by layer. Read it in ten minutes, then follow the path for your role.

Who this book is for

It is written for the **CTO, VP of Engineering or lead architect at a mid-size company**, roughly 50 to 1,000 engineers. Companies of that size rent almost every layer of the stack they ship on. They are big enough to own or control a few layers, such as the release path and the product's own interface, and small enough that a wrong bet on a layer hurts.

You do not need to be a kernel engineer. Every technical term is explained the first time it appears. You do need to answer for three things: the **security** of what runs on your customers' and your staff's machines, the **cost** of keeping it running, and the **value** it creates for the business. The book is organized around those three words.

Two kinds of readers will get less from it. Hyperscale cloud providers already own most of their stack. Early startups should rent nearly every layer for now, with one exception: the path that builds, signs and updates their own product, which Chapter 14 argues they should control from the start. Their path is Part I, then Chapter 14; the rest of Part III can wait.

Three companies, three sets of stakes

Abstract advice about "the stack" is easy to agree with and hard to use. So every chapter tests its ideas on the same three companies. None is a real client; each is a composite of a common kind of business.

> **THE SOFTWARE MAKER**
>
> It ships a desktop app and a web app to thousands of business customers, on Windows, macOS and sometimes Linux. Its stakes are in the layers it chooses to bundle: a browser engine inside its app, a language runtime, an installer, an update channel. Each one it ships becomes its problem on every customer's machine, and each one it leaves to the OS vendor becomes a roadmap it does not control.

THE REGULATED COMPANY

It works in finance, health care or government, and it must prove what runs on its machines, not just believe it. Its stakes are evidence and response: which code can reach the kernel, which boot keys its laptops trust, how fast a patch moves from vendor to fleet, and whether it can show an auditor the answer on one page.

THE DEVICE FLEET OPERATOR

It is a hotel group or a retailer, running thousands of identical machines in the field: point-of-sale terminals, check-in kiosks, self-service screens. The machines live for years and sit in places with no engineer nearby. Its stakes are uptime and the cost of a site visit. One bad update or one missed certificate date hits every location at once.

The value map

Here are the nine 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	Value
Software maker	How many copies of a browser engine do we ship, and how fast can we patch each one? **Ch 7**	What does bundling a runtime per app cost our customers in memory, disk and update work? **Ch 11**	Which few layers are worth owning because they make our product different? **Ch 15**
Regulated company	Who can change code inside our kernel, and how do we prove it to an auditor? **Ch 2**	What does patch lag across all our vendors cost us in exposure and incident work? **Ch 11**	How far can identity, signing and attestation prove what a person actually saw and approved? **Ch 4**
Device fleet operator	Which boot certificates and firmware support dates will expire under our fleet, and when? **Ch 1**	Which drivers and devices can we standardize to shrink the update list and the site visits? **Ch 3**	Should we control the OS image and update path for our kiosks, and what would it take? **Ch 14**

What you will be able to do

- **Name every owner** between your code and your customer's screen, layer by layer, on Windows, Linux and macOS.

- **Fill in a Pixel Bill of Materials** for your own product: one row per component, with who writes it, who pays, who can ship an update to your machines and when its support ends.
- **Put a number on the seams**: the memory, update channels and patch lag that come from renting many layers from many owners.
- **Make a rent, fund, fork or build call** for each layer, tied to one business metric, and defend it to your board.
- **Plan incident response by owner**, so that when a flaw crosses layers, you already know whom to call and what each one can fix.

Reading paths by role

Role	Start with	Then	Keep for reference
CTO or VP Engineering	Prologue, Chapter 10 (Why the Winners Won)	Part III: Chapters 11 to 15	Part I, one chapter per layer, as questions come up
Platform lead or architect	Part I in order: Chapters 1 to 8	Chapters 11, 12 and 14	Appendix C, the full comparison table
Security lead	Chapters 1, 2 and 4	Chapters 7 and 13 (AI Moves the Seams)	The Security lens box at the end of every chapter

Part II (Chapters 9 and 10) tells how the three systems came to be. It is the best place to understand why each vendor behaves as it does, and it reads well on its own.

The Pixel Bill of Materials

A factory knows every part in its product, who makes it and what it costs. Software teams rarely know the same about the screen their customers look at. The **Pixel Bill of Materials** is that list for the whole stack, from the processor to the lit pixel. Each chapter in Part I adds its rows, and Chapter 15 hands you a blank one. The chapter tables have seven columns.

Column	What it asks
Component	The real part, by name: a firmware, a kernel, a driver, a library, an engine.
Who writes it	The company or project whose engineers change the code.
Who pays	Who funds that work: a vendor through its prices, sponsors, or you.
Who can update it	Who can ship a new version to your machines. This is often not the same as who writes it, and neither is always the party that signs it; chapters say so where the difference matters.
Support clock	When updates stop, or when a certificate or key expires.
Reachable from untrusted input?	Can a file, web page, network packet or image from outside reach this code? If yes, its flaws are attack paths.
Your choice	**Rent:** use the vendor's layer as shipped. **Fund:** pay for or staff an open layer you do not control; it buys influence, not control. **Fork:** carry your own modified copy. **Build:** write your own. Combinations are allowed.

In this book, to **own** a layer means to fork it or build it: the code is yours. To **control** a layer means you decide its composition, signing and rollout, though you did not write it, for example an OS image you compose from vendor parts and pin, or your release pipeline running on rented build servers. Configuring or pinning is control, not a fork. The release path is the main example: in every bill its choice reads **Control (build only the trust-critical pieces)**, meaning rented build servers, with your own pipeline and your own signing keys. When the book speaks of a vendor, it says the vendor **supplies** or **controls** a layer, as in "the OS vendor controls the consent prompt". Chapter 15's full bill adds three columns to the seven: **patch time when AI finds it first**, **tested recovery time** and **accountable owner**, the person who answers for the row.

Here is one worked row, from Chapter 2, for the endpoint security agent that a regulated company runs on its laptops.

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Endpoint security agent with a kernel component	The security vendor	You, by subscription	The vendor, often many times a day, with no change window	Your contract term	Yes: it inspects untrusted files and traffic by design	Rent; require staged rollout and a plan to move out of the kernel

Read the row from left to right and a risk appears that no single column shows: a party you pay, but do not control, can change code at the highest privilege level on every machine you own, without waiting for you. That is not a reason to drop the product. It is a reason to negotiate how it updates. In July 2024, a faulty update to one such agent stopped about 8.5 million Windows devices, by Microsoft's count. Chapter 2 tells that story.

The one idea to carry

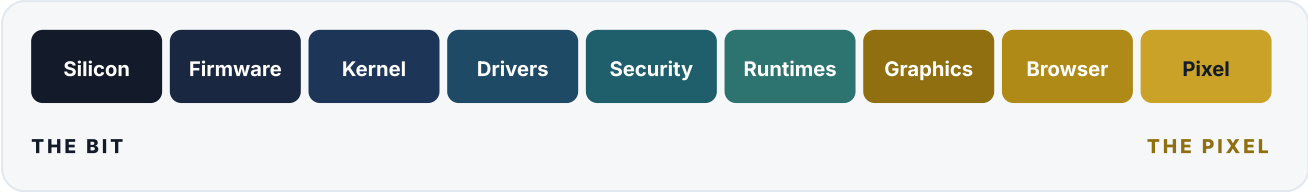
Every layer has an owner. Owning more layers does not make a product better or safer by itself, and renting a layer is not a failure. What ownership changes is narrower and more useful to know.

> *Owning a layer changes who can fix it, and how fast.*

Keep that sentence next to every decision in this book. When you ask "should we own this?", ask instead: when this layer breaks, who will fix it, how long will it take, and can we live with that answer?

PROLOGUE

One Key Press, and One Question



You press the letter a. A moment later a small shape of light appears on the screen. It feels like one event. It is a relay race through every layer of the machine, and almost every runner in that race works for someone else.

This book follows that race. It starts at a key switch the keyboard controller encodes as bits, and ends at the pixel, one lit dot on a display. Between the two sit firmware, a kernel, drivers, an input system, an application runtime, a text engine, a compositor and a graphics processor. Windows, Linux and macOS each run the race in their own way, and each answer was shaped by decades of history and business.

The reason to follow it is not curiosity. Every layer in that race has an owner, a roadmap and a price. When a layer changes, your product changes with it, whether or not you were ready. A CTO who can see the whole race can decide where to stand in it. The Reader's Brief names the three companies that test that decision in every chapter.

The same key, three paths

Here is the letter a on each system, from the keyboard to the light. The names are real components you can look up. The shape is the same on all three; the owners are not.

Step	Windows	Linux (Wayland desktop)	macOS
Key report	The keyboard sends a USB HID report that carries the modifier state (Shift, Ctrl, Alt) and usage code 0x04, the key position named "a and A"; the operating system then applies the keyboard layout to decide which character it becomes		
Kernel input	kbdhid.sys, kbdclass.sys	usbhid, evdev	IOHIDFamily
Event to the app	win32k, WM_KEYDOWN and WM_CHAR	libinput, xkbcommon, Wayland key event	WindowServer, NSEvent
Text	DirectWrite	HarfBuzz shapes, FreeType draws	Core Text
Compose the screen	Desktop Window Manager	Mutter or KWin	WindowServer, Core Animation
GPU and display	Direct3D, WDDM driver	Mesa, DRM and KMS	Metal
Light	The new frame is shown at the next display refresh		

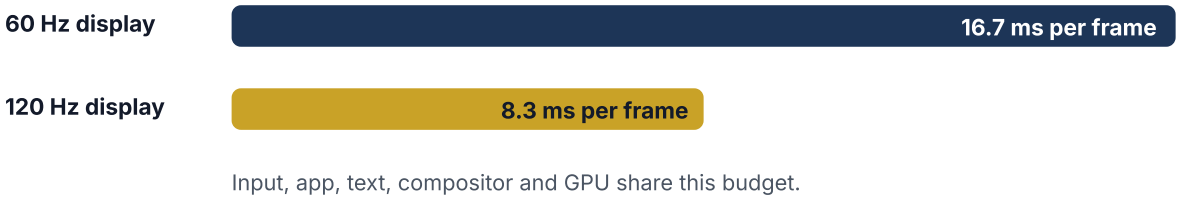
The table shows the common composed path. It is not the only one: a full-screen app can sometimes skip the compositor and hand its frames straight to the display hardware, through direct scanout, hardware overlays or, on Windows, independent flip.

HOW IT GOT THIS WAY

A PS/2 keyboard raised a hardware interrupt, IRQ 1, each time a key changed. A USB keyboard works the other way. USB defines an "interrupt" transfer type, but it is the host that polls the keyboard, at an interval the device states in its endpoint descriptor, often 1 to 10 milliseconds. The USB host controller then interrupts the processor when data arrives. So a modern key press mixes polling and interrupts, and the stack keeps traces of both designs. That is the first lesson of this book: the stack is history you can still see.

FIGURE P.1

The time budget for one frame



Input, app, text, compositor and GPU share this budget.

One second divided by the refresh rate. Every layer in this book spends part of it, and a miss costs the user a whole extra frame.

> *Every layer between the bit and the pixel has an owner. Owning a layer changes who can fix it, and how fast.*

The Pixel Bill of Materials

The book builds a list of every layer on that path, one chapter at a time, and calls it the **Pixel Bill of Materials**. Here is a simplified preview; the full columns are in the Reader's Brief.

Layer	Who writes it	Who pays for it	Who can update it	Your choice
Text shaping	An open project with a small core team	Mostly sponsors	Each app or OS that ships a copy	Rent or fund
Browser engine	Upstream project, set mostly by one vendor	That vendor	The runtime vendor, or each app that bundles it	Rent, and plan for the seam
Your release path	Your team, on rented build servers	You	You	Control (build only the trust-critical pieces)

For each row a company has four honest choices. It can **rent** the layer: use the vendor's layer as shipped. It can **fund** it: pay for or staff an open layer it does not control, which buys influence, not control. It can **fork** it: carry its own modified copy. Or it can **build** it: write its own. None of these wins by default. Owning a layer does not make a product better; what it moves is the repair: whose engineers fix a defect, and on whose schedule. Each chapter in Part I adds its rows to the bill, and Chapter 15 hands you a blank one for your own company.

BUSINESS IMPLICATIONS

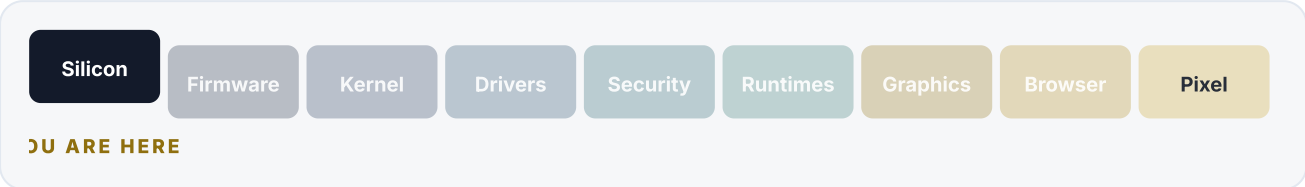
Security. The owners on this path play different parts: some write the code, some sign it, and some can ship an update to your customers' machines. Know which is which, how fast each one patches, and which of them run inside the kernel.

Cost. Each layer brings its own update channel, test cycle and copies in memory. That seam tax is real money, and few companies add it up.

Value. Owning a layer is a business decision, not a language choice. Chapter 15 gives the tests for choosing the few layers worth owning or controlling, and for renting the rest on good terms.

CHAPTER ONE

Silicon and Boot



Before your operating system runs a single line, three other owners have already run theirs: the chip maker, the firmware vendor and the company that holds the boot signing keys. You did not choose most of them, and a mid-size fleet cannot, in practice, patch their firmware or re-sign their boot keys itself. Yet when one of them fails, it is your fleet that stops booting. This chapter is about the code that runs first, and about the calendar it quietly keeps.

Rings, modes and the memory map

A processor does not trust every instruction equally. It runs in **privilege levels**, and the hardware decides which level may touch which part of the machine. On Intel and AMD processors (the x86 family) these levels are called **rings**, numbered 0 to 3, and a lower number means more privilege. Ring 0 holds the most privilege the processor gives an operating system, though a hypervisor, the firmware's System Management Mode and platform protections can still limit it. Ring 3 holds the least, and code there may do only what more privileged code permits. In practice, Windows, Linux and macOS on x86 use just two of them: ring 0 for the kernel, ring 3 for applications.

Arm processors, which power every Mac Apple now sells and most phones, use **exception levels** instead. EL0 runs applications, EL1 runs the operating system kernel, EL2 runs a hypervisor (the layer that hosts virtual machines) and EL3 runs the most trusted firmware. Here the numbering runs the other way: a higher level means more privilege. The idea is the same. Code at a more privileged level can generally reach the state of the levels beneath it, and a hypervisor and the firmware add privilege domains above the kernel.

The second protection is the **memory management unit**, or MMU. Ordinary applications work with virtual addresses, not physical ones. Each process sees its own private map, called **virtual memory**, and the MMU translates every address on the fly using tables the kernel

writes. Microsoft's driver documentation puts the result plainly: because each application's address space is private, one application cannot modify another's data. Kernel code gets no such private map, and Chapter 2 shows what that costs.

The GPU, a second computer

The graphics processor, or GPU, is not a helper chip. It is a second computer with its own instruction set, its own scheduler and its own firmware. A discrete graphics card also has its own memory, called video memory; an integrated GPU, built into the processor package as in many laptops, shares main memory with the CPU. In the hardware-accelerated path, the CPU does not draw your pixels. It writes commands into a buffer, and the GPU runs them on its own clock and rasterizes the result. A CPU can still draw through a software rasterizer, the fallback Chapter 3 describes. Its driver, written by the GPU vendor, is one of the largest pieces of code on any desktop, and Chapter 3 and Chapter 6 return to it.

On a PC with a discrete graphics card, the GPU works fastest on data in its own video memory, so programs usually copy data there, although the GPU can also read mapped system memory across the bus, more slowly. Microsoft's Direct3D 12 documentation names the two cases: a discrete adapter has local video memory and non-local system memory, while an integrated, or UMA, adapter has a single segment, system memory. Apple built its own machines around the shared model. When it introduced the M1 chip on 10 November 2020, Apple described a **unified memory architecture**: one pool of memory shared by the CPU, GPU and other engines, so they can use the same data without copying it between pools. That is a hardware choice with software effects. Integrated GPUs on Windows and Linux PCs share memory too. What Apple's control of both the chip and the operating system adds is that its frameworks for graphics, machine learning and video can assume that one design on every Mac it sells.

Firmware: the code before the code

Firmware is the program stored on a chip on the motherboard. It wakes the hardware, checks memory, finds a disk and hands control to an operating system loader. On PCs and servers today that firmware follows the **UEFI** standard (Unified Extensible Firmware Interface), maintained by the UEFI Forum, which counts more than 250 member companies.

Few PC makers write their own UEFI firmware. Most license a core from an **independent BIOS vendor**: chiefly AMI, Insyde and Phoenix. The PC maker adds its own code on top, then ships the result. So a single firmware flaw can have three owners: the chip maker's reference code, the BIOS vendor's core and the PC maker's build. Each one may need to ship a fix before your machine is safe.

Secure Boot is the UEFI feature that decides what may run next. The firmware holds a small set of key databases, set at the factory. Microsoft's hardware guidance names them: the signature database (`db`) lists signers that are trusted; the revoked database (`dbx`) lists signers and images that are not; the Key Exchange Key database (`KEK`) holds the keys allowed to update those two lists; and the Platform Key (`PK`), set by the PC maker, controls the KEK. Before the firmware runs a boot loader, it checks the loader against `db` and `dbx`. When Secure Boot is enabled and enforcing, a loader runs only if a key in `db` signed it or its hash is listed in `db`, and only if `dbx` has not revoked it.

A **TPM** (Trusted Platform Module) is a small security chip, or a protected function inside the processor, that stores keys and records measurements of what booted. Windows 11 requires TPM 2.0 and firmware that is UEFI and Secure Boot capable. In November 2020 Microsoft announced **Pluton**, a security processor built into the CPU with AMD, Intel and Qualcomm. Its design first shipped in the Xbox One in 2013. On Windows PCs, Pluton first acts as a TPM, and its firmware is updated through Windows Update, which moves part of the hardware trust root onto Microsoft's patch schedule.

Boot step	Windows PC	Linux PC	Mac with Apple silicon
First host firmware stage relevant to OS boot	UEFI firmware from the PC maker, usually on an AMI, Insyde or Phoenix core	The same UEFI firmware	Boot ROM, fixed in the chip at manufacture
Who signs the next stage	Microsoft (Windows boot manager)	Microsoft signs `shim`; the distribution signs GRUB and the kernel	Apple (LLB, then iBoot)
Key and policy store	UEFI `db`, `dbx`, `KEK`, `PK`	UEFI keys, plus the Machine Owner Key list in `shim`	LocalPolicy, signed by the Secure Enclave
Hardware key holder	TPM 2.0 or Pluton	TPM 2.0	Secure Enclave, with its own boot ROM and sepOS
Who ships firmware fixes	The PC maker; some via Windows Update	The PC maker; some via the Linux Vendor Firmware Service	Apple, inside macOS updates

The UEFI firmware is not the very first code on a PC. Security processors inside the chip, such as Intel's Converged Security and Management Engine and AMD's Platform Security Processor, start earlier from their own ROM and can check the firmware before the main

cores run it. They belong to the chip maker, and they are part of its row in your bill.

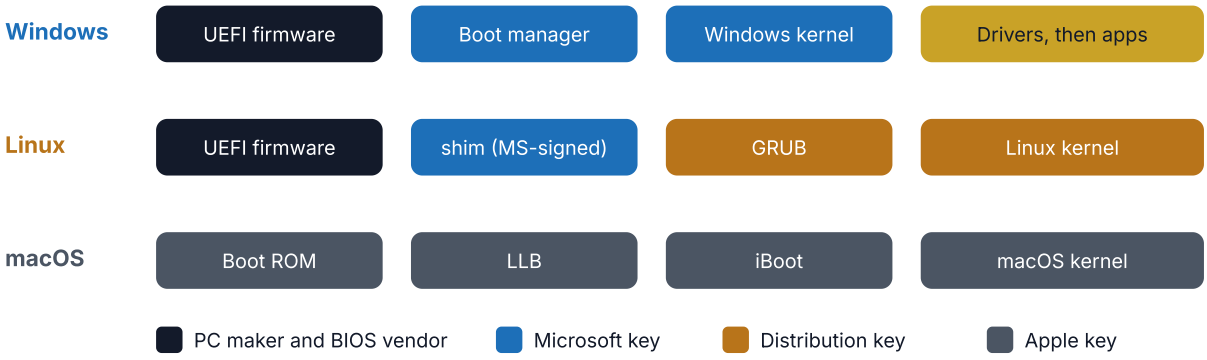
On Linux, the chain has one more link. Because most PCs ship trusting Microsoft's keys, many mainstream Linux distributions ship a small first-stage loader called **shim**, which Microsoft signs through its third-party UEFI certificate. Shim carries the distribution's own key and uses it to check the next loader, usually GRUB, which then checks the kernel. That design keeps Microsoft out of each distribution's release cycle: Microsoft signs shim rarely, and the distribution signs everything after it. Shim is the common path, not the only one. An owner can enroll their own keys in the firmware, boot a **unified kernel image (UKI)**, a single signed UEFI file holding the kernel and its start-up data, directly from firmware that trusts its signer, or turn Secure Boot off.

Apple supplies and controls the whole path on its own chips. The Boot ROM is laid down when the chip is made and cannot be changed. Apple calls it the hardware root of trust. It holds the public key of Apple's root certificate authority (the private key never leaves Apple) and checks the next stage, the Low-Level Bootloader (LLB). On a Mac with Apple silicon, LLB loads the LocalPolicy and checks iBoot, which checks the macOS kernel. The **Secure Enclave**, a separate security processor on the same chip, boots its own small operating system, sepOS, from its own boot ROM, and it signs the LocalPolicy file that records which security mode the owner chose.

FIGURE 1.1

The chain of trust, and who controls each link

Each box checks the signature of the box to its right before it runs it.



On the common shim path, a Linux PC trusts a Microsoft signature before it trusts its own distribution; owner-enrolled keys, a directly signed UKI or Secure Boot turned off avoid that. A Mac in Full Security, the default, trusts only Apple. A Windows PC trusts its PC maker first and Microsoft second.

HOW IT GOT THIS WAY

The original PC firmware, the BIOS, came with the IBM PC in 1981 and was built for a 16-bit processor. It survived for decades through clever extensions: network cards, for example, carried option ROMs that added network boot, standardized by Intel's Preboot Execution Environment (PXE) specification, version 2.1 in 1999. But it had no standard way to check that the code it started was signed. In 1998 Intel started the Intel Boot Initiative to design a new boot model for its Itanium servers. The result was published as EFI, ending at version 1.10. In 2005 the industry formed the Unified EFI Forum, Intel contributed EFI 1.10, and the specification became UEFI, open to any processor. Secure Boot arrived in UEFI 2.3.1, and Microsoft made it a hardware requirement for Windows 8 certified PCs in 2012. That decision is why Microsoft's certificates sit inside almost every PC sold since, and why Linux had to invent shim to boot on them.

Microcode: the patchable processor

Modern processors are partly programmable. Many complex instructions run as small internal programs called **microcode**, and the chip maker can ship new microcode to change how the processor behaves after it is sold. A microcode update is not stored permanently in the processor; it is loaded into the processor at every boot, either by the firmware or by the operating system.

Each system gets it differently. On Linux, the kernel loads microcode early from the initial RAM disk, using files that distributions package from Intel and AMD; the kernel's own documentation warns that loading it late, while the system runs, is risky and has been off by default since kernel 5.19. On Windows, microcode arrives mostly inside firmware updates from the PC maker, and Microsoft has also published Intel microcode packages itself, starting with KB4090007 in March 2018. On a Mac, Apple delivers firmware inside macOS updates, so the owner never sees a separate step.

When one flaw crosses four owners

On 3 January 2018, Google Project Zero and several academic teams published **Meltdown** and **Spectre**. Project Zero had reported them to Intel, AMD and Arm on 1 June 2017. The flaws were not bugs in any operating system. They lived in **speculative execution**, the processor's habit of guessing ahead to save time, which left traces that an attacker could measure to read memory it should never see.

No single owner could fix it. The chip makers shipped microcode. The kernel teams changed how memory is mapped: Linux added page table isolation (KPTI), and Windows and macOS shipped their own changes. Compiler teams added mitigations. Browser vendors cut the

precision of JavaScript timers; Mozilla reduced `performance.now()` to 20 microseconds and disabled `SharedArrayBuffer` in Firefox 57.0.4. Intel's first Spectre microcode caused unexpected reboots and, on 22 January 2018, Intel told customers to stop deploying it until fixed versions arrived. A CTO who wanted to know "are we patched?" had to ask four different owners, on four different schedules.

Firmware flaws travel the same way. In July 2020 Eclypsium disclosed **BootHole** (CVE-2020-10713), a buffer overflow in how GRUB2 reads its configuration file. Because the vulnerable GRUB was signed and trusted, an attacker could use it to bypass Secure Boot on Windows and Linux machines alike. The fix needed new GRUB and shim builds from every distribution, plus revocation updates to `dbx` on every machine. In December 2023 Binarly presented **LogoFAIL**: flaws in the image parsers that UEFI firmware uses to show the maker's logo at boot. They affected firmware from all three main BIOS vendors, AMI, Insyde and Phoenix. A crafted logo image runs too early for Secure Boot to stop it. The only fix is a new firmware build from each PC maker, and each one shipped it on its own schedule.

> *A firmware flaw has three owners and a fix has three release dates. Your fleet is patched on the latest one.*

The expiry calendar

Certificates expire. The Microsoft certificates that Secure Boot has trusted since 2011 were issued with fixed end dates, and those dates have now arrived. This is not a flaw. It is a calendar, and a fleet that does not keep one finds out the hard way.

DATED BOX: SECURE BOOT 2011 CERTIFICATES (AS OF NOVEMBER 2026)

Microsoft's support article "Windows Secure Boot certificate expiration and CA updates" lists three expiring certificates and their four replacements:

- **Microsoft Corporation KEK CA 2011**, which authorizes updates to `db` and `dbx`: expired June 2026; replaced by Microsoft Corporation KEK 2K CA 2023.
- **Microsoft UEFI CA 2011**, which signs third-party boot loaders (including Linux shim) and option ROMs: expired June 2026; replaced by Microsoft UEFI CA 2023 and Microsoft Option ROM UEFI CA 2023.
- **Microsoft Windows Production PCA 2011**, which signs the Windows boot loader: expired October 2026; replaced by Windows UEFI CA 2023.

Microsoft states that devices without the 2023 certificates will keep starting and running, and will keep receiving normal Windows updates, but will no longer be able to receive new security protections for the early boot process.

Nothing breaks on the day. The machine still boots. What stops is the ability to receive future boot-level fixes, such as new `dbx` revocations after the next BootHole. A fleet that misses the change looks healthy and is quietly frozen at its old trust list.

EXAMPLE: THE FLEET OPERATOR'S EXPIRY CALENDAR

A hotel group runs 4,000 check-in kiosks and point-of-sale terminals from three hardware makers, bought over six years. Its platform team keeps one shared calendar with a row per item: the Secure Boot certificates in each model's firmware, the firmware support end date from each maker, the TPM firmware version, the Linux shim or Windows boot loader in each image, and the code-signing certificates of its own apps. Each row has a date, an owner and a test: one kiosk of each model in a lab, updated a quarter ahead. When a maker has stopped shipping firmware for an older model, the row says so, and that model goes on the replacement budget. The calendar costs a few days a year. A missed row costs a site visit to every location.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
CPU microcode	Intel, AMD (Apple for its own chips)	Built into the chip price	The PC maker in firmware, or the OS vendor at boot; the chip maker writes and signs it	Chip maker's support list per processor family	Indirectly: Spectre showed web pages could probe it	Rent; track it per model
UEFI firmware	AMI, Insyde or Phoenix core, plus PC maker code	Built into the device price	The PC maker, directly or through Windows Update or LVFS	PC maker's support term for that model	Yes: LogoFAIL used a boot logo image	Rent; buy models with long firmware support
Secure Boot certificates	Microsoft (and the PC maker for the PK)	Microsoft	Microsoft through Windows Update, or the PC maker through firmware; Microsoft signs the new keys	The 2011 set expired in June and October 2026	Indirect: firmware parses the signed db and dbx update payloads, and dbx is how BootHole-class loader bugs are revoked	Rent; keep the calendar
Linux `shim` and GRUB	Open source projects, built by each distribution	Distributions and their sponsors	Your distribution; Microsoft signs shim	The distribution's release support	Yes: BootHole used a GRUB config file	Rent from a distribution; fund if you depend on it
Apple Boot ROM and iBoot	Apple	Built into the Mac price	Apple only (Boot ROM cannot be updated)	Apple's macOS support for that Mac	Limited, and inside Apple's control	Rent

SECURITY LENS

Who can change it? The chip maker, the BIOS vendor, the PC maker and the holder of the boot keys, usually Microsoft. On a Mac, only Apple. You can add trust without replacing the Platform Key: on Linux, shim lets a machine owner enroll a Machine Owner Key (MOK) and boot kernels and modules signed with it. What a mid-size fleet cannot do in practice is patch the firmware or re-sign the vendors' keys.

How fast can it be fixed? Slowly, and on the schedule of the slowest owner. A firmware fix must pass from the BIOS vendor to the PC maker to your machines, model by model. Some parts, such as the Apple Boot ROM, can never be patched at all; the system is designed so that later stages guard against its limits.

What does AI change here? AI-assisted binary analysis makes it cheaper to search shared firmware code, such as the logo parsers LogoFAIL hit, but a fix still travels from BIOS vendor to PC maker to your machines, model by model. Plan for flaws that are found faster than that chain can ship.

BUSINESS IMPLICATIONS

Security. The regulated company must be able to prove what booted. That means TPM measurements, Secure Boot on, and a current `dbx` on every machine. An auditor who asks "how do you know your laptops trust the 2023 certificates?" deserves a report, not a guess.

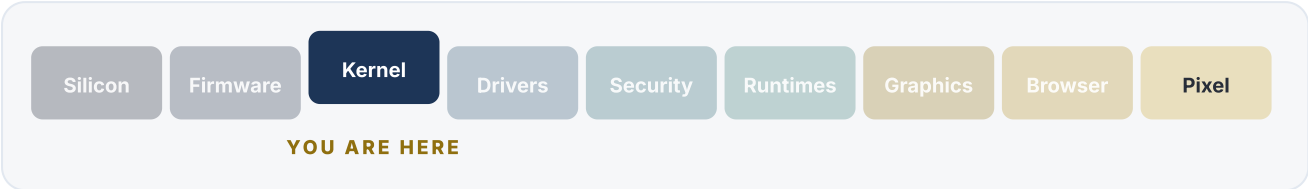
Cost. For the device fleet operator, the price of a kiosk includes its firmware support term. A model that is cheaper by a small amount but loses firmware updates three years earlier costs more over a seven-year life, because it ends in a truck roll or an early replacement.

Value. The software maker cannot own this layer and should not try. Its value is in knowing it: testing its installer and drivers on machines with the new certificates, and telling customers early when a boot change will touch them.

This quarter: put every certificate, firmware support date and boot key your machines depend on into one calendar, with an owner per row.

CHAPTER TWO

The Kernel



On 19 July 2024, a configuration file for a security product reached Windows machines around the world, and about 8.5 million of them stopped at a blue screen. Airports, hospitals, hotels and stores went dark the same morning. The file was not part of Windows, and none of the affected companies wrote it. It was rented code running in the one place where a single mistake stops everything: the kernel.

What a kernel does

The **kernel** is the core of the operating system: the program that runs with full hardware privilege, in ring 0 on x86 or EL1 on Arm, as Chapter 1 explained. Everything else asks it for help. It does four main jobs.

- **Processes and threads.** A **process** is a running program with its own private memory. A **thread** is one line of work inside it. The kernel creates them, tracks them and ends them.
- **Virtual memory.** The kernel writes the page tables that the memory management unit uses, so each process sees its own private map and cannot read another's data.
- **Scheduling.** A machine has a few processor cores and thousands of threads. The **scheduler** decides which thread runs on which core, and for how long, many times a second.
- **System calls and drivers.** Programs reach files, the network, the screen and every device through **system calls**, the kernel's front door. Behind that door sit **drivers**, the code that talks to each piece of hardware.

The kernel is also a wall. Microsoft's driver documentation states the rule that governs this chapter: all code in kernel mode shares a single address space, so a kernel driver is not isolated from other drivers or the operating system, and if it crashes, the entire operating system crashes. An application usually fails alone. Kernel code lacks that process isolation: a

bug in it can crash or corrupt the whole system. Linux can sometimes survive a kernel fault, called an oops, by stopping only the task that hit it, but the kernel then marks itself as tainted, and its state may no longer be reliable.

Three kernels, three designs

Kernel designers argue about how much code belongs behind the wall. A **monolithic** kernel puts the core and all its services, including file systems, networking and drivers, into one program in kernel mode. A **microkernel** keeps only a small core in kernel mode and runs services as separate programs outside it. That can contain a failing service, at the price of more crossings of the wall; how much those crossings cost depends on the design, so neither model is safer or slower by definition. The three systems in this book chose differently.

Linux is monolithic, with **loadable modules**: drivers and file systems can be loaded into the running kernel when needed, but once loaded they run with full privilege like the rest of it. **XNU**, the macOS kernel, is in Apple's own words a hybrid kernel combining the Mach kernel developed at Carnegie Mellon University with components from FreeBSD and a C++ driver API called IOKit. The Mach part brings microkernel ideas such as message passing; the BSD part brings the Unix process model and networking; all of it runs together in kernel mode. **Windows NT** is also usually called a hybrid: a small kernel at the bottom, an "executive" of services above it, and drivers alongside, all in kernel mode, with the windowing and graphics core (`win32k`) in the kernel too.

Topic	Windows (NT kernel)	Linux	macOS (XNU)
Design	Hybrid: kernel plus executive, drivers in kernel mode	Monolithic with loadable modules	Hybrid: Mach plus BSD plus IOKit
Who controls the source	Microsoft (closed)	Upstream maintainers, led by Linus Torvalds (GPLv2)	Apple (source published for parts of Darwin)
Scheduler model	Preemptive, 32 priority levels (0 to 31), round-robin within a level	EEVDF since Linux 6.6 (October 2023), replacing CFS	Mach threads, guided by app quality-of-service classes
Third-party code in the kernel	Common: drivers, security agents, anti-cheat	Common: modules from hardware and security vendors	Restricted: kernel extensions deprecated since 2019
Where outside code is being moved	User mode, under the Windows Resiliency Initiative	No single policy; upstream prefers drivers merged into the tree	System Extensions and DriverKit, in user space
Rust in the kernel	Parts of `win32k` (GDI regions) since 2023 builds	Supported since 6.1 (December 2022); Rust Binder driver in 6.18 (November 2025); experiment declared concluded in December 2025	Not announced

Scheduling: who gets the processor

Every kernel must answer the same question thousands of times a second: which thread runs next? The answers shape how a system feels.

Windows uses priorities. Microsoft documents 32 levels, from 0 to 31. The scheduler gives time slices in turn to all ready threads at the highest level, and a higher-priority thread that becomes ready stops a lower one at once. Linux has used a fairness model since 2007. In version 6.6, released on 29 October 2023, it replaced the Completely Fair Scheduler with **EEVDF** (Earliest Eligible Virtual Deadline First), which tracks how much processor time each task is owed and gives the next turn to the one with the earliest deadline among those that are eligible. macOS asks applications to label their work with **quality-of-service**

classes, from user-interactive work such as animations to background work such as indexing, and the system uses those labels to set scheduling, processor and disk throughput and timer behavior.

A CTO rarely tunes a scheduler directly. But the scheduler decides whether the kiosk's touch response stays smooth while a virus scan runs, and whether a trading desk's market feed waits behind a backup job. Knowing which model your platform uses tells you which knobs exist.

Rented code in ring 0

A running kernel also hosts code its owner did not write. Graphics drivers, storage drivers, network drivers, virtualization tools, backup agents, anti-cheat software and endpoint security products all load kernel components. Each one is rented code with full privilege. When it is wrong, it is wrong for the whole machine.

The July 2024 outage is the clearest case. CrowdStrike's own technical summary says that at 04:09 UTC on 19 July 2024 it released a sensor configuration update to Windows systems, a file known as Channel File 291. The update had a logic error. The Falcon sensor's kernel component read it and crashed, and with it the operating system. CrowdStrike reverted the file at 05:27 UTC, 78 minutes later. By then, machines that had received it were stuck: they crashed at every boot before they could fetch the fix, and many needed a person at the keyboard. Microsoft estimated that 8.5 million Windows devices were affected, less than one percent of all Windows machines.

Three lessons follow. First, the content that triggered the crash was not a new driver; it was a data file read by a driver already trusted. A review process for code alone would not have caught it. Second, the speed of the vendor's update channel, usually a strength, carried the fault everywhere at once. Third, the recovery was manual, and so its cost scaled with the number of machines and the number of locations.

FIGURE 2.1

Where third-party security code runs, before and after



Apple moved outside code out of its kernel first, because it controls the whole platform. Microsoft is following, with partners who need time to re-engineer. Linux leaves the choice to each vendor and each distribution.

Moving rented code out of the kernel

Apple moved first. At its 2019 developer conference, with macOS Catalina 10.15, it introduced **System Extensions** and **DriverKit**, which let developers build drivers and security tools that run in user space instead of in the kernel. Apple's developer page states that kernel extensions have been deprecated and that, starting with macOS Big Sur (2020), macOS no longer loads kernel extensions that use deprecated interfaces by default. Those that still load cannot load on demand: they need an administrator's approval and a restart, and on a Mac with Apple silicon the owner must first lower the startup security policy to Reduced Security in recoveryOS and allow kernel extensions from identified developers. Security products moved to the **Endpoint Security** interface and network filters to **Network Extension**. Developers need an entitlement from Apple to use them. Apple could do this because it controls the hardware list, the kernel and the developer program.

Microsoft's path is harder, because Windows runs on hardware from thousands of makers and a large security industry built its products in the kernel. In November 2024, four months after the outage, Microsoft announced the **Windows Resiliency Initiative**. Its stated plan includes new Windows capabilities "that will allow security product developers to build their products outside of kernel mode", so that antivirus and endpoint protection can run in user mode, as apps do. In June 2025 Microsoft said it would deliver a private preview of the Windows endpoint security platform to partners in its Microsoft Virus Initiative the following month. At Ignite in November 2025 it said the Windows Endpoint Security Platform API was in preview with those partners, who were helping to make it

generally available in 2026. On 17 November 2026, at Ignite, Microsoft announced that the platform "is reaching general availability on Windows 11". General availability of the platform is not the same as your vendor's product running on it: each security vendor still has to ship a version that works outside the kernel. Treat it as a migration in progress: ask each of your kernel-level vendors whether its product runs on the new platform yet, and when.

Linux has no equivalent single move. Its upstream maintainers prefer drivers merged into the main kernel tree, where they are reviewed and kept current; out-of-tree modules are the vendor's burden. Security tools increasingly use kernel interfaces that limit what outside code can do, but each vendor chooses its own design, and each distribution its own policy.

> **HOW IT GOT THIS WAY**
>
> All three kernels descend from ideas of the 1960s to the 1980s. Unix began at Bell Labs in 1969, when Ken Thompson and Dennis Ritchie started work on a little-used PDP-7 computer. Its process model and file model shaped both BSD and Linux. At Carnegie Mellon University, Project Mach ran from 1985 to 1994 and explored a microkernel that could host Unix as a service. NeXT built its operating system on Mach and BSD, and when Apple bought NeXT, that design became Darwin and XNU; Mac OS X 10.0 shipped on 24 March 2001. At Microsoft, a team led by Dave Cutler, hired from Digital Equipment Corporation in 1988, began a new portable kernel that shipped as Windows NT 3.1 in July 1993. NT 4.0 and Windows 2000 ran on that line, and every consumer Windows since Windows XP has too. Linux started on 25 August 1991, when Linus Torvalds told a newsgroup he was writing a free operating system that was "just a hobby". Decades on, those three decisions still decide who can change your kernel.

Rust enters the kernel

Most kernel code is written in C, a language that leaves memory safety to the programmer. A large share of serious security flaws are memory errors that C allows and safer languages prevent. **Rust** checks memory use when the code is compiled. It has no garbage collector and no virtual machine, so it can work at the kernel level.

Linux merged initial Rust support in version 6.1, released on 11 December 2022. It was infrastructure, not drivers: the scaffolding that let later drivers be written in Rust. In December 2025, at the kernel maintainers summit, the project declared the Rust experiment concluded and successful; the patch that removed the "experimental" label said, in the words of its author Miguel Ojeda, that Rust is here to stay. He added that this does not mean

everything works for every configuration, architecture and toolchain. The first large Rust driver had already landed: Linux 6.18, released on 30 November 2025, added a Rust version of Binder, the inter-process communication driver used by Android.

Microsoft took a narrower route. In July 2023, Windows Insider build 25905 included `win32kbase_rs.sys`, a new Rust implementation of GDI regions, part of the graphics core inside the Windows kernel. Microsoft said it would continue to increase the use of Rust in the kernel. The component later reached Windows 11 version 24H2, and it gave an instructive lesson. In 2025 Check Point Research reported that a malformed EMF+ metafile, a Windows graphics format, could crash 24H2 machines through that Rust code. Rust did its job: it detected an out-of-bounds access and stopped instead of corrupting memory. But in the kernel, stopping means a system crash. Microsoft rated it a moderate denial-of-service issue and fixed it in 2025. Memory safety blocks whole classes of memory corruption, so a flaw that might have been a takeover became a crash. It does not stop logic errors, and it does not make kernel code harmless.

> *Memory safety can turn a takeover into a crash. In the kernel, a crash is still an outage.*

Why most companies rent the kernel, and tune it

Almost no company should write its own kernel. The three in this book carry decades of hardware support, security response and compatibility that no mid-size team can match. The useful question is not "own or rent?" but "how much do we tune, and how much outside code do we let in?"

Tuning is not ownership in this book's sense, which means a fork or a build, but it is real control at low cost. A team can choose a long-term-support kernel line instead of the newest, set scheduler and memory options for its workload, remove modules it does not need, and decide which third-party drivers and agents may load. On Windows, policy tools can block unsigned or unwanted drivers. On macOS, the device management profile decides which System Extensions are allowed. On Linux, a team can compile the distribution's kernel source with its own configuration, which is the most control of the three and also the most work to keep current. It becomes a fork only when the team starts carrying its own patches.

EXAMPLE: A KERNEL INVENTORY

A regulated bank lists every component that loads into the kernel on its standard laptop and server images: the drivers from each hardware maker, its endpoint security agent, its disk encryption, its virtualization tools. For each it records the vendor, how updates arrive (and whether content updates are staged or pushed to all machines at once), whether the vendor has a user-mode version on the way, and how a broken machine is recovered without a visit. The list has fewer than twenty rows. It becomes the bank's answer when an auditor asks, "who can change code in your kernel, and on what schedule?"

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Windows NT kernel	Microsoft	Microsoft, through licenses	Microsoft only	The Windows release's servicing dates	Yes: files, network, fonts and graphics reach kernel code	Rent; control which drivers load
Linux kernel	Upstream maintainers and thousands of contributors, many paid by companies	Contributing companies; distributions	Your distribution, or you if you build your own kernel	The long-term-support line you pick	Yes	Rent from a distribution; tune; fund if critical
XNU	Apple	Built into the Mac price	Apple only	Apple's macOS support for that Mac	Yes	Rent
Endpoint security agent in the kernel	The security vendor	You, by subscription	The vendor, often many times a day	Your contract	Yes: it inspects untrusted files and traffic by design	Rent; demand staged rollout and a user-mode plan
Hardware drivers (GPU, storage, network)	Hardware makers	Built into the hardware price	The maker, through the OS vendor or its own tool	The maker's support for that device	Often: network and graphics drivers parse outside data	Rent; standardize models to shrink the list

SECURITY LENS

Who can change it? The OS vendor, and every vendor whose driver or agent you allow to load. On Windows and Linux that list is often longer than anyone has written down. On macOS it is short by design.

How fast can it be fixed? The OS vendor fixes its kernel on a monthly or release schedule. A kernel-level vendor can push changes in minutes, which is good for new threats and dangerous for new mistakes. A fix that requires booting a machine that cannot boot is only as fast as the person who must reach it.

What does AI change here? AI tools help translate kernel C into Rust and search third-party drivers for memory errors, but the 24H2 crash shows that safer code still needs review and an owner on call. The bigger gain is less outside code in the kernel, which no tool provides.

BUSINESS IMPLICATIONS

Security. For the regulated company, every component in its kernel is an audit question. Keep the inventory, require each kernel-level vendor to stage its updates, and ask for a written plan to move to user mode as Windows and macOS make that possible.

Cost. The device fleet operator paid for July 2024 in site visits, not licenses: a machine that cannot boot cannot heal itself. Budget for remote recovery (out-of-band management, recovery images, encryption keys you can reach) before you need it. It costs far less than one morning of dark check-in desks.

Value. The software maker gains by keeping its own code out of the kernel. A product that runs in user space, through supported interfaces, is easier to ship on all three systems, survives OS updates and is not a liability on its customers' machines.

First step: list everything that loads into your kernel, and for each row write who can push a change and how a broken machine gets back up.

CHAPTER THREE

Drivers, Storage and Network



The kernel decides who runs. Drivers decide what the hardware actually does. Every frame your customer sees, every record your system saves and every packet it sends passes through code written by a company that is neither you nor, very often, the operating system vendor. This chapter is about those owners, and about the one data path that some companies choose to own outright.

A **driver** is the code that translates between the kernel and one piece of hardware: a disk controller, a network card, a graphics processor, a fingerprint reader. Because it touches hardware directly, a driver has traditionally run inside the kernel with full privileges, so one bad driver can take down the whole machine. The three systems have answered that risk in three very different ways, and each answer tells you whose engineers repair a driver bug, and on whose schedule.

Three driver models, three bargains

Windows runs on hardware from thousands of manufacturers, so Microsoft had to give outsiders a stable, documented way into the kernel. The older **Windows Driver Model (WDM)** is powerful but notoriously hard to get right. Microsoft layered the **Windows Driver Frameworks (WDF)** on top: the Kernel-Mode Driver Framework (KMDF) for drivers that must run in the kernel, and the User-Mode Driver Framework (UMDF) for drivers that can run as ordinary processes. Microsoft published the WDF source code on GitHub in March 2015, a few months before Windows 10 shipped.

Stability came with a gate. Starting with Windows 10, version 1607, a fresh install with Secure Boot on will not load a new kernel-mode driver unless Microsoft's own developer portal has signed it, and, as Microsoft's driver documentation still states as of November 2026, the portal account must have at least one extended validation (EV) code-signing certificate registered to it. In other words, a third party writes the driver, but Microsoft decides whether it may run in the kernel at all.

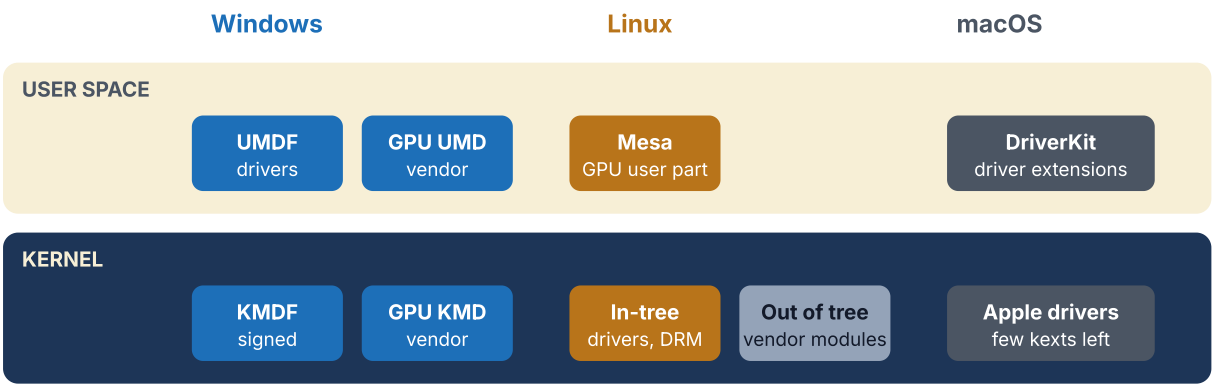
Linux made the opposite bargain. The kernel promises applications a stable system-call interface, but it promises drivers no stable interface inside the kernel. The kernel document `stable-api-nonsense`, written by Greg Kroah-Hartman, says plainly that Linux "does not have a binary kernel interface, nor does it have a stable kernel interface". Internal interfaces change whenever developers fix bugs or improve designs. The advice to hardware makers is direct: "get your kernel driver into the main kernel tree." A driver that lives **in-tree** is updated by whoever changes the interface it depends on. A driver kept **out of tree** breaks on its owner's schedule, not yours.

macOS took a third path, available because Apple controls most of its own hardware. **DriverKit** moves many drivers out of the kernel into user space as "driver extensions", and Chapter 2 tells how Apple retired the kernel extensions they replace. A crashing driver extension can be restarted like any other process.

Question	Windows	Linux	macOS
Driver interface	WDM, with KMDF and UMDF on top; stable across releases	No stable in-kernel interface; drivers belong in the kernel tree	DriverKit in user space; kernel extensions deprecated
Who writes most drivers	Hardware vendors	Vendors and kernel developers, upstream	Apple, plus vendors for add-on devices
Who lets a driver run	Microsoft signs kernel drivers (Windows 10, 1607 and later)	The distribution that builds the kernel, and the machine owner	Apple, through code signing and approval of system extensions
Fix path for a driver bug	Vendor ships it; Windows Update may carry it	Upstream patch, then each distribution's kernel	Apple's OS update, or the vendor's extension

FIGURE 3.1

Where a driver runs



Code in the dark band can crash the machine. Code in the light band can be restarted.

The same job, placed three ways. Windows splits drivers by framework, Linux keeps most drivers in the kernel tree, and macOS moves them out to user space where it can.

The GPU driver: a separate owner

The graphics driver deserves its own line in any inventory, because on two of the three systems it belongs to a company that is neither you nor the OS vendor.

On Windows, Microsoft introduced the **Windows Display Driver Model (WDDM)** with Windows Vista. WDDM splits the graphics driver in two: a user-mode display driver that does most of the heavy work, and a streamlined kernel-mode driver. It also brought **timeout detection and recovery**, which lets Windows restart a hung display driver without a reboot. The GPU maker writes both halves and ships them on its own schedule.

On Linux, the kernel part of most graphics drivers lives in the kernel tree under the Direct Rendering Manager (DRM), and the user part lives in **Mesa**, an open project that implements OpenGL and Vulkan for many GPUs. The major exception has been NVIDIA, whose driver was long closed. In May 2022 NVIDIA published open-source GPU kernel modules, dual-licensed GPL and MIT, and its later drivers moved fully to them for supported GPUs. They are still distributed outside the kernel tree.

On macOS with Apple silicon, Apple designs the GPU and writes its driver. One owner, one update channel. That is simpler, and it also means no other party can ship you a fix.

SIDEBAR: THE SILENT FALLBACK

Windows and Linux keep a documented software path for drawing when the GPU path fails. Windows has WARP, a software rasterizer that Microsoft documents for use "when a video driver is not available, or is not working correctly". Linux has Mesa's llvmpipe, a software rasterizer that uses the CPU. On macOS, Apple documents a software renderer only for its OpenGL framework, which falls back to it unless an app asks for hardware alone; OpenGL has been deprecated since macOS 10.14, and Apple documents no such fallback for Metal.

The risk is that the fallback is quiet. After a driver update, a new GPU in a cloud image, or a changed virtual machine template, an application can end up drawing on the CPU. The screen still works. Nothing crashes, so error monitoring sees nothing. What changes is cost: CPU use rises, frames take longer, battery drains faster, and a fleet of cloud desktops or kiosks needs bigger machines.

The fix is cheap. Record which renderer and which data path a machine actually uses, as a metric, not a log line, and alert when it changes after an update.

File systems: where your data is shaped

A **file system** decides how bytes become files on a disk: how they are laid out, how a crash is survived, and whether the system can tell when data has silently rotted.

Windows uses **NTFS** for nearly everything, including the boot volume. Microsoft announced the **Resilient File System (ReFS)** in January 2012 and shipped it with Windows Server 2012. ReFS keeps checksums on its metadata and, optionally, on file data, so it can detect corruption and, with mirrored storage, repair it while the volume stays online. As of November 2026 Microsoft's ReFS documentation still lists it as not bootable and not supported on removable media, and no released Windows version boots from it, so NTFS remains the general-purpose default.

Linux offers a choice, which is both a strength and a decision you must make. **ext4** was declared stable in Linux 2.6.28, released on 24 December 2008, and remains a common default. **XFS**, which originated on Silicon Graphics' IRIX systems, is a multi-threaded journaling file system built for large files and large file systems. **Btrfs** was merged into the mainline kernel in January 2009, during the 2.6.29 cycle, and adds copy-on-write, checksums for data and metadata, and snapshots. Each distribution picks a default, so two Linux machines in the same fleet can store data in different ways.

macOS moved to the **Apple File System (APFS)**, the default for Macs from macOS High Sierra (10.13) in 2017. Apple designed it for flash storage, with native encryption as a capability of the file system, copy-on-write metadata, cheap file clones and read-only

snapshots of a whole volume. On a Mac with Apple silicon or a T2 chip the internal volume stays encrypted even with FileVault off, but the user's credentials do not gate it: Apple says the key is then protected only by a hardware identifier in the Secure Enclave. **FileVault** is the setting that ties the key to the user's password, and so decides whether data is protected at rest against someone who holds the Mac.

Feature	Windows	Linux	macOS
Default file system	NTFS	ext4 or XFS, by distribution; Btrfs on some	APFS
Snapshots	Volume Shadow Copy; ReFS file-level snapshots on Server 2022 and later	Btrfs, or the volume manager	APFS snapshots
Encryption at rest	BitLocker, below the file system	dm-crypt, below the file system	Built into APFS; FileVault ties the key to the user's password

The network path: sockets, rings and programs in the kernel

Almost every program that talks over a network still uses an idea from 1983. The **sockets** interface arrived with 4.2BSD Unix, and it treats a network connection much like a file you can read and write. Linux and macOS use BSD sockets directly. Windows adopted the same idea through **Windows Sockets**, or Winsock, whose version 1.1 specification is dated 20 January 1993 and is modeled on the Berkeley interface.

Sockets have a cost: each read or write is a system call, a trip into the kernel and back. For servers handling millions of operations a second, those trips add up. In Linux 5.1, released in May 2019, Jens Axboe's **io_uring** added a different model: the application and the kernel share two ring buffers in memory, one for requests and one for completions, so a program can submit many operations with few system calls. It began with storage and grew to cover networking. It also added a large new surface inside the kernel, which matters for the security lens below.

Apple went the other way for app developers. At WWDC 2018 it introduced **Network.framework** as "a modern alternative to sockets": a higher-level interface that handles connection setup, TLS encryption and network changes by default, and that Apple's own URL loading system is built on. You give up some control and gain behavior Apple keeps current.

The third idea is **eBPF**: small programs that you load into the Linux kernel at run time, which the kernel checks with a verifier before running them on events such as an arriving packet. Teams use eBPF for fast packet filtering, load balancing and observability without writing a kernel module. Microsoft announced the **eBPF for Windows** project in May 2021, an open-source effort that reuses existing eBPF tools on Windows. As of November 2026 its own repository still describes it as a work in progress, so it is not yet a replacement for the Linux feature set.

Who controls the data path

The **data path** is the route a request takes from the network card to your code and back, and from your code to the disk. On a default system, every step belongs to someone else: the NIC vendor's driver, the kernel's network stack, the OS vendor's TLS library, the file system, the storage driver. For most companies that is right: these are mature layers, patched by large teams.

Ownership starts to pay when the data path is the product, when it is your largest cost, or when you need a fix faster than any vendor will ship one. Cloudflare is a public example: it built **Pingora**, its own proxy in Rust, to replace the older proxy software in front of its network. Chapter 14 tells that story in full.

Cloudflare built the proxy, where its business logic and costs concentrate. It did not write its own kernel, drivers or file system. It owned that narrow layer and rented the broad layers beneath it.

> *A driver is a contract you never signed. Know who wrote it, who lets it run, and who will ship the fix.*

HOW IT GOT THIS WAY

Windows Vista moved most graphics driver code out of the kernel with WDDM, because display drivers had been a leading cause of crashes. Linux never offered drivers a stable interface at all, and asked vendors to join the tree instead. The Windows signing gate is described above, and Apple's move away from kernel extensions is told in Chapter 2. Three systems, one problem, three owners: Microsoft as gatekeeper, the kernel community as maintainer, Apple as the vendor of nearly everything.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
GPU driver (WDDM, Mesa, Apple)	GPU vendor, Mesa developers, or Apple	The vendor, funded by hardware sales	The vendor; on Linux, also the distribution	Vendor drops older GPUs on its own schedule	Yes: shaders and content from web pages and documents reach it	Rent; pin and test versions
Network card driver and stack	NIC vendor and OS vendor	Vendors	OS vendor or distribution	OS support life	Yes: every packet from the network	Rent
File system (NTFS, ext4, APFS)	OS vendor or kernel community	OS vendor; on Linux, contributing companies	OS vendor or distribution	OS support life	Yes, through removable media and disk images	Rent; choose deliberately on Linux
Proxy or load balancer in front of your service	An open project, a vendor, or you	You, through license, cloud bill or staff	Whoever you chose	Your own, if you build it	Yes: it is the first thing the internet touches	Rent, fund or build, by scale

SECURITY LENS

Who can change it? Kernel-mode driver code runs with the highest privilege the operating system has, and most of it is written by hardware vendors; user-mode drivers (UMDF, DriverKit extensions, the user-mode half of a GPU driver) usually take down only their own process when they fail. On Windows, Microsoft controls which kernel drivers may load. On Linux, the distribution decides which drivers ship in its kernel. On macOS, Apple writes most drivers and approves the rest. Your company can choose hardware and versions, and little else.

How fast can it be fixed? As fast as the slowest owner in the chain. A network driver flaw on Linux is fixed upstream, then picked up by each distribution, then installed by you. A GPU driver flaw on Windows waits for the GPU vendor. Fleets that pin driver versions for stability must also own the job of moving the pin.

What does AI change here? AI-assisted fuzzing makes it cheaper to search drivers and in-kernel interfaces such as io_uring and eBPF, which parse complex input at full privilege, so more of their flaws will surface, for vendors and attackers alike. For a fleet that pins driver versions, that puts the weight on moving the pin: have a tested way to roll a new driver to every model in days.

BUSINESS IMPLICATIONS

Security. A **device fleet operator** running thousands of kiosks has more driver owners than it thinks: GPU, touch screen, card reader, receipt printer, network. Most of them run at least partly in the kernel, and each comes from a different company, with a different patch rhythm. List them, by model and version, before an incident forces the list.

Cost. A **software maker** shipping a graphics-heavy desktop app pays for GPU driver diversity in support tickets and in silent software fallbacks that make the app feel slow on some machines. A **regulated company** choosing a Linux file system pays later if it picks one without checksums for records it must prove intact. Both costs are avoidable with a test matrix and a renderer metric.

Value. Owning the data path pays only where it is the product or the biggest bill, as the proxy was for Cloudflare. For most companies the valuable act is narrower: choose drivers and file systems on purpose, measure the path the hardware really takes, and keep the fix path short.

Before the next driver update: list every kernel-level driver in your fleet with its owner and version, and alert when a machine falls back to a software path.

CHAPTER FOUR

Security, Identity and the Trusted Pixel



Every approval in your company ends with a person looking at a screen and pressing a button. A payment is released, a contract is signed, an administrator grants a privilege. The whole chain assumes the screen showed the truth. This chapter explains how each operating system decides who may do what, and then asks the question almost nobody puts in a risk register: who vouches for the pixel the person actually saw?

A **trust boundary** is a line in the system where the rules change: on one side code may do something, on the other it may not. Kernel and user space form the first boundary, covered in Chapter 2. This chapter covers the boundaries built on top of it: between users, between applications, between a background service and the person at the keyboard. Each system draws them differently, and its vendor controls the drawing.

Trust boundaries on each system

Windows gives every process an **access token**, a kernel object that says which user it runs as, which groups it belongs to and which privileges it holds. Every file, registry key and other object carries an **access control list (ACL)** that says which users and groups may do what. When a program opens a file, the kernel compares the token with the list. **User Account Control (UAC)**, introduced with Windows Vista, gives an administrator two tokens: a filtered one for daily work and a full one used only after approval. By default, the approval prompt appears on the **secure desktop**, a separate desktop that, in Microsoft's words, only Windows processes can access. Ordinary programs cannot draw on it or send it input.

Linux starts from the Unix model: a user ID, a group ID and permission bits on each file, with the superuser, root, allowed everything. Since Linux 2.2 the kernel has divided root's power into separate **capabilities**, such as the right to bind a low network port or to load a kernel module, so a service can hold only the ones it needs. On top of that, **Linux Security Modules** add mandatory rules that even root must obey. **SELinux**, released by the US

National Security Agency in December 2000, labels every process and file and checks each access against a central policy. **AppArmor**, merged into the mainline kernel in Linux 2.6.36 in October 2010, confines programs by file path and is simpler to write.

macOS keeps the Unix model underneath and adds Apple's own layers. **System Integrity Protection (SIP)**, introduced with OS X El Capitan in 2015, stops even the root user from changing protected system files and folders. The **App Sandbox** confines an application to the resources it declares. Those declarations are **entitlements**: signed key-value pairs inside an app's code signature that grant specific rights. Finally, **Transparency, Consent and Control (TCC)** asks the user, or an administrator through device management, before an app may use the camera, the microphone, protected folders or screen recording.

Boundary	Windows	Linux	macOS
Identity of a process	Access token	User and group IDs, plus capabilities	User ID, plus code signature and entitlements
Who may touch an object	ACL on each object	Permission bits; SELinux or AppArmor policy	Permission bits; sandbox profile
Rules even admin cannot skip	Protected processes, driver signing	SELinux or AppArmor in enforcing mode	System Integrity Protection
Asking the person	UAC on the secure desktop	Desktop-specific prompts (polkit, portals)	TCC prompts
Service manager	Service Control Manager	systemd on most distributions	launchd
Sign-in framework	Windows Hello and credential providers	PAM	Touch ID matched in the Secure Enclave, through Authorization Services and LocalAuthentication; PAM for `sudo` and some services

Services and identity

Most code on a machine runs with nobody watching. A **service manager** starts that code, restarts it, and decides which identity it runs as. On Windows that is the **Service Control Manager (SCM)**, which runs each service under an account such as LocalSystem or a dedicated virtual account. On most Linux distributions it is **systemd**, which Lennart

Poettering introduced in April 2010 and which can apply capabilities, private file systems and other limits to each service from one unit file. On macOS it is **launchd**, which Poettering himself cited as the model for systemd's idea of starting services on demand. The service manager is where least privilege is either configured or forgotten.

Sign-in is the other half of identity. On Linux and other Unix systems, **Pluggable Authentication Modules (PAM)**, first proposed by Sun engineers in Open Software Foundation RFC 86.0 in October 1995, let administrators change how users prove who they are without changing each program. Microsoft announced **Windows Hello** for Windows 10 in a Windows Experience blog post on 17 March 2015, using face, iris or fingerprint sensors. On Macs, **Touch ID** matching runs in the **Secure Enclave**, a separate processor; Apple says the fingerprint data is encrypted and protected with a key available only to the Secure Enclave. In all three cases the OS vendor controls the sign-in screen, the sensor path and the prompt the person reads.

The Trusted Pixel

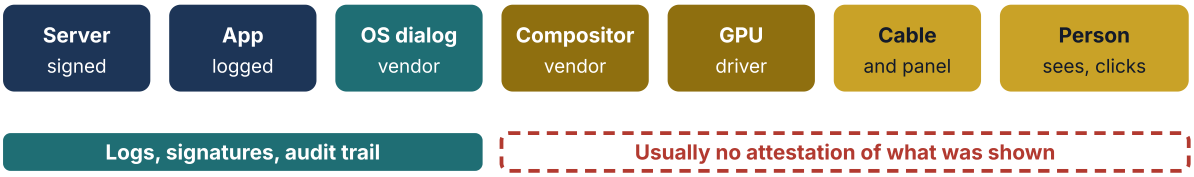
The principle is not new. Signature designers call it **WYSIWYS**, "what you see is what you sign", the title of a 1998 paper by Peter Landrock and Torben Pedersen. This book applies it to every approval, not only signatures: **what the person saw must be what the system did**. Every consent dialog, payment confirmation, sign-in prompt and electronic signature rests on it. The system proves a great deal about the transaction. It signs the request, logs the user, timestamps the click. On most systems it proves far less about the screen.

Less is not nothing. **Android Protected Confirmation**, added in Android 9 in 2018 as an optional feature on supported devices, hands control to a hardware-protected user interface that shows a short statement full screen. If the person approves, the app signs that statement with a key that can sign nothing else. Google's documentation says a positive answer indicates "with high confidence that the user has seen the given text, even if the Android framework (including the kernel) was compromised." Do not build on it alone. It needs dedicated hardware and works only where the device maker supports it. In January 2024 an Android change marked the internal code behind it as deprecated, citing its cost to device makers and low adoption by developers; in May 2024 that change was reverted as an "inadvertent partial deprecation". As of November 2026 Google's public `ConfirmationPrompt` reference carries no deprecation notice, but a feature with that history belongs in a design only as an extra where a device offers it. The real gap is narrower and still wide. As of November 2026, Windows, macOS and Linux desktops offer third-party apps no general facility like it: the Windows secure desktop protects system prompts such as UAC, not your payment screen. Wherever no such facility is used, your logs prove the transaction, not what the person saw.

"Trusted pixel" also covers five different threats, and each needs a different control. Mixing them is how teams buy a control for one threat and believe it covers the others.

FIGURE 4.1

The consent path, and where the evidence stops



The click travels back with proof. The picture traveled forward without it.

Overlays, screen capture, agents and deepfakes all work in the right-hand half.

A consent leaves strong evidence on the left of this path and, unless a protected confirmation screen is used, little on the right. The right-hand half belongs to the OS vendor, the GPU vendor and the display maker.

Threat model	What goes wrong	Who supplies the control	Control
1. Transaction integrity	The data signed is not the data the person meant	You, inside your regulator's rules	Dynamic linking: bind amount and payee into the signed request
2. Trusted UI	Another program draws over, or imitates, the approval screen	The OS vendor	Protected surfaces where they exist (secure desktop, protected confirmation, overlay filtering); for third-party desktop apps, which have none, a second-channel confirmation
3. Identity and deepfakes	The face or voice on the screen is not who it seems	You, through process	Out-of-band call-back; approvals no single channel can complete
4. Capture and privacy	What the screen shows is copied, stored or exported	The OS vendor builds it; you set the policy	Capture policy: display affinity, Recall policy, export rules
5. Delegated agents	Software acting with the user's authority reads and accepts the approval	You, through identity and permissions	Separate agent identities, scoped permissions, no agent approves value alone

Threat model 1: transaction integrity. The question is whether the data the system signs is the data the person meant to approve. In the European Union, the strong customer authentication rules under the second Payment Services Directive (PSD2), set out in Commission Delegated Regulation (EU) 2018/389, require **dynamic linking**. Article 5 requires that "the payer is made aware of the amount of the payment transaction and of the payee", that the authentication code is specific to that amount and payee, and that any change to either invalidates the code. Article 5(2) goes further: providers must adopt security measures which ensure "the confidentiality, authenticity and integrity" of the amount and payee and of "the information displayed to the payer throughout all of the phases of the authentication". Malware that shows one payee and submits another is therefore a failure of controls the rule already requires. The narrower gap remains: the rule sets the duty, but no standard mechanism lets you prove afterwards what the screen showed. The authentication code proves which amount and payee were authenticated, not what the person saw. Digital identity follows the same pattern. Regulation (EU) 2024/1183, known as eIDAS 2.0, entered into force on 20 May 2024 and establishes the **European Digital Identity Wallet**. The European Commission states that member states must provide wallets to citizens by the end of 2026, weeks after this November 2026 edition, so check

each member state's wallet status on the day you design for it. A wallet shows attributes and signature requests on a phone screen, and the person's approval there is what the relying party trusts. **Control:** bind amount, payee and identity into what is signed, verify that binding on the server, and for high-value actions show the bound data again on a channel the requesting device does not control.

Threat model 2: trusted UI. The question is whether another program can draw over the approval or fake it. If it can, the person sees one thing and taps another. On Android this attack is common enough that Google documents it as **tapjacking** and gives developers a flag, `filterTouchesWhenObscured`, to ignore touches while another window covers the view. On Windows the defense for administrator consent is the secure desktop, where other programs cannot draw. On supported Android devices, Protected Confirmation goes further and shows the approval text through a hardware-protected interface that a compromised operating system cannot alter. The OS vendor supplies each of these controls. An application can ask for them; it cannot build them. On Windows, macOS and Linux desktops, a third-party app has no protected surface for its own approval screen. **Control:** use the protected surface the platform offers wherever it exists; for third-party approvals on the desktop, send the final confirmation to a second channel, such as a separate device, a signed out-of-band approval in your own app on a second device, or a call-back, and treat Protected Confirmation as an extra where a device supports it, not a dependency.

Threat model 3: identity and deepfakes. The question is whether the face or the voice on the screen belongs to who it seems. In January 2024 a finance employee in Hong Kong joined a video call in which the chief financial officer and the other colleagues were deepfakes, built, police said, from publicly available video and audio. The employee made 15 transfers totaling HK$200 million, about US$25.6 million. In May 2024 the engineering firm Arup confirmed it was the victim. The call software worked exactly as designed. **Control:** call back on a known number before any transfer above a threshold, and require approvals that no single channel can complete.

Threat model 4: capture and privacy. The question is who can copy, store or pass on what the screen shows. Windows offers **display affinity**: `SetWindowDisplayAffinity` with `WDA_EXCLUDEFROMCAPTURE`, supported since Windows 10 version 2004, keeps a window off screen captures. Microsoft is explicit that this is not a security guarantee, since nothing stops a camera pointed at the screen. Microsoft announced **Recall** on 20 May 2024, with its first Copilot+ PCs, a Windows feature that saves regular snapshots of the screen so a user can search what they have seen. After security researchers and the public criticized the design, Microsoft delayed it and changed it. Its 27 September 2024 post describes the new design: Recall is always opt-in, snapshots are always encrypted, the storage services run in a virtualization-based security enclave, and access requires Windows Hello. Microsoft's management documentation says Recall is disabled and removed by default on managed

devices, and that administrators decide whether users may enable it and, in the European Economic Area, whether they may export their snapshots. The OS vendor can decide that what a machine shows is also stored; the protection of that store is the vendor's design, and the policy around it is yours. The sidebar below covers the regulatory edge: for some companies, showing a screen can be an export. **Control:** set capture policy on purpose (Recall, screen recording, remote support tools), mark sensitive windows with display affinity, and treat screen sharing of controlled technology as an export compliance decision.

Threat model 5: agents acting under delegated authority. The question is no longer who is on the screen but what is at the keyboard. On 22 October 2024 Anthropic released **computer use** in public beta: a model that looks at screenshots, moves the cursor, clicks and types. OpenAI released **Operator**, a browser agent, as a research preview on 23 January 2025. Such agents act with the user's privileges and read the same pixels a person reads, so a consent dialog designed for a human can be accepted by software on the human's behalf. Nobody is impersonated: the agent holds real, delegated authority, and a prompt injected into a page or document can steer how it uses it. **Control:** give agents their own identities and narrowly scoped permissions, log their actions under those identities, and keep payments and signatures out of reach of any agent acting alone.

> *The OS vendor or desktop project, not you, controls the consent pixel. That is an own-or-rent decision, whether or not anyone made it on purpose.*

For most companies the right choice is still to rent that layer, because the secure desktop, the sign-in screen and protected confirmation are built by the OS vendor, not the app. Renting well means knowing it is rented. Put the consent screens your business depends on in your inventory, name which of the five threat models each one faces, and give each the control that matches.

SIDEBAR FOR REGULATED COMPANIES: SCREEN SHARING CAN BE AN EXPORT

Under the US Export Administration Regulations (EAR), technology can be exported without anything leaving the building. 15 CFR 734.13 defines a **deemed export**: releasing controlled technology or source code to a foreign person inside the United States. 15 CFR 734.15(a)(1) says technology is "released" through "visual or other inspection by a foreign person of items that reveals 'technology' or source code subject to the EAR". A screen share, a remote support session or a shared virtual desktop can be such an inspection.

This applies only to technology and source code that are subject to the EAR and require a license for that person. It does not make every screen share an export. But for a defense supplier, a chip designer or a research group, the pixel is a border crossing, and screen-sharing policy belongs with the export compliance team.

HOW IT GOT THIS WAY

The NSA released SELinux to the open source community in December 2000, and the debate over whether to accept it led to Linux Security Modules, a general hook that let SELinux, and later AppArmor in 2010, plug into the kernel. Windows Vista, released to consumers on 30 January 2007, added UAC and the secure desktop: administrators now did daily work with a filtered token, and approved their own elevation on a screen other programs could not touch. Apple added SIP in 2015, so that not even root could alter the operating system. Each step moved the line between trusted and untrusted code. None of them attests what the display shows.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Consent and elevation prompts (UAC, TCC, polkit)	OS vendor or desktop project	OS vendor	OS vendor or distribution	OS support life	Indirect: overlays, spoofed UI and agents target it; it parses no untrusted files	Rent; confirm high-value actions on a second channel
Sign-in and biometrics (Windows Hello, PAM, Touch ID)	OS vendor and sensor maker	OS and hardware vendors	OS vendor	OS and device support life	Partly: presentation attacks on sensors	Rent
Mandatory access policy (SELinux, AppArmor, sandbox profiles)	Distribution or OS vendor; you for custom rules	Vendor, plus your staff time	You, for your own policy	Yours, for custom policy	Indirectly: it limits what a compromised process can reach	Rent the framework; build your own policy
Your approval screens for payments and signatures	You	You	You	Yours	Yes: overlays, capture, agents	Build; bind amount and payee into what is signed, and confirm on a protected screen or second channel

SECURITY LENS

Who can change it? The OS vendor controls tokens, capabilities, SIP, the secure desktop and every system prompt. Administrators control the policy that runs on top. Application teams own only their own screens, and those screens are drawn by the vendor's compositor.

How fast can it be fixed? A flaw in a system prompt or a sandbox waits for the OS vendor's release. A weak approval design in your own product can be fixed in your next release, which is why the binding of amount, payee and identity into the signed request is the part to own.

What does AI change here? Deepfakes make a face or voice on a video call cheap to forge, and computer-use agents can read and accept a consent screen with the user's privileges. Both break the assumption behind threat models 3 and 5, that a known human looked at the pixel, so approvals of value need a step that neither a forged face nor an agent can complete alone.

BUSINESS IMPLICATIONS

Security. A **regulated company** releasing payments on the strength of a video call or a screen approval is trusting a layer it does not own. Require call-back on a known number for any transfer above a threshold, and make sure agents running on staff machines cannot approve payments or signatures alone.

Cost. A **software maker** that builds its own signing or approval flow pays to design it twice: once for people and once for the agents that will operate it. A **device fleet operator** running self-service kiosks must decide whether screen capture, overlays and remote support tools are allowed on machines that take payments, and enforce it through policy, not habit.

Value. The value is trust that survives a dispute. A company that can show which amount and payee were authenticated, and that the person confirmed them on a second channel, has evidence a screenshot cannot give. It still cannot prove what the screen showed unless the confirmation ran on a protected display, so do not promise more than that in your terms.

First action: list every approval your business depends on, and for each one write down who controls the screen it appears on and which of the five threats it faces.

CHAPTER FIVE

Runtimes and Languages



Your developers do not write for the kernel. They write for a runtime: a library, a virtual machine or an interpreter that sits between their code and the operating system. That runtime decides how memory is handled, how fast a security fix reaches your customers, and who is allowed to ship it. Most companies picked their runtimes years ago, one project at a time, and never wrote the choice down.

What a runtime really is

A **runtime** is the code that must be present on the machine for your program to run, even though your team did not write it. It starts the program, hands out memory, reports errors, talks to the operating system on the program's behalf, and often cleans up after it.

Runtimes come in three rough sizes. The smallest is a **language library**, such as the C runtime: startup code, memory allocation, file and string functions. In the middle is a **managed runtime**, such as the Java Virtual Machine or the .NET Common Language Runtime (CLR). It adds a **garbage collector**, a part that finds memory the program no longer uses and frees it, and usually a **just-in-time compiler**, which turns portable bytecode into machine code while the program runs. The largest is a **whole platform** that carries a language engine, a standard library and often a browser engine, such as Node.js or Electron.

The size matters less than who supplies it. Every runtime has a vendor or a community that writes it, a support clock that ends, and an update channel. A defect in the runtime is a defect in your product, and you can only fix it as fast as that channel allows.

The native platforms: three operating systems, three contracts

Each operating system offers a native runtime contract, the set of libraries an application can count on being there.

Windows offers Win32, the C interface to the system that dates back decades, and the C runtime beneath it. Since Windows 10, Microsoft ships the **Universal C Runtime** (UCRT) as part of the operating system. Microsoft's documentation states that on Windows 10 and 11 the system copy is always used, even when an application carries its own local copy. On top sits .NET: the older .NET Framework, which is a Windows component, and the modern cross-platform .NET, which ships on its own schedule.

Linux offers the kernel's system calls and, almost always, the **GNU C Library** (glibc), which implements the C standard and POSIX. **POSIX** is the portable operating system interface standard; its first edition, IEEE 1003.1, was published in 1988. On Linux, the distribution controls glibc: Red Hat, Canonical, Debian or your own build team decides which version you get and when it is patched.

macOS offers libSystem, Apple's C and POSIX layer, and above it the Cocoa frameworks and the Objective-C and Swift runtimes. Apple announced Swift on 2 June 2014. With Swift 5 in 2019 the language became **ABI stable**, meaning compiled code from different compiler versions can work together, and the Swift runtime started shipping inside the operating system from macOS 10.14.4. Since then Apple, not the app, updates it.

Question	Windows	Linux	macOS
Base C runtime	Universal CRT, part of Windows 10 and later	glibc (musl on some small distributions)	libSystem
Native app framework	Win32, with .NET on top	None from the kernel; GTK or Qt chosen per app	Cocoa, with Swift and Objective-C
Managed runtime of the vendor	.NET CLR	None by default	None with a garbage collector; Swift uses reference counting
Who patches the base runtime	Microsoft, through Windows Update	The distribution vendor	Apple, through OS updates
Can an app override it?	Not the UCRT on Windows 10 and 11	Yes, with bundles or containers	Rarely, and only for older Swift targets

Runtimes that travel: the JVM, Node.js and Python

Three runtimes promise the same code on every operating system. Each keeps the promise by bringing its own world with it.

The **Java Virtual Machine** runs bytecode on any machine that has a JVM. Today many vendors build their own JVM from the open OpenJDK code, so a company picks a JVM vendor as well as a Java version. **Node.js** wraps a JavaScript engine with system access; the project states that its long-term support (LTS) lines typically receive critical fixes for a total of 30 months. **Python** is an interpreter with a vast library ecosystem; under its release policy (PEP 602) each version is supported for about five years.

Compare those clocks with .NET, where, as of November 2026, Microsoft's support policy gives LTS releases three years of free support and standard-term support (STS) releases two years, that is, 24 months. A team that runs Java, Node.js, Python and .NET runs four support calendars, four security feeds and four upgrade projects that never line up.

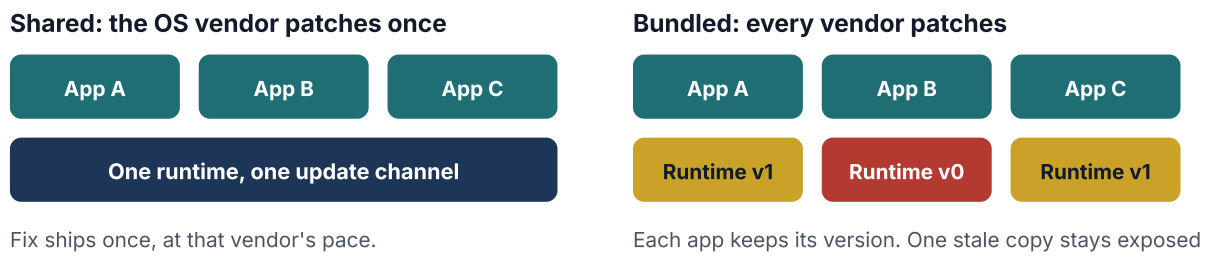
Why every app now ships its own world

Bundling solves a real problem. A team that ships its runtime inside the app knows exactly which version of that runtime its customers run, and a system update cannot swap it out. Bundling pins the dependency, not the platform: OS interfaces, drivers and security policy below the bundle still change with every system update. Electron says so in its own documentation: by embedding Chromium and Node.js into its binary, it lets one JavaScript codebase run on Windows, macOS and Linux. Self-contained .NET apps, Java apps with a private JVM, and Python tools with a frozen interpreter all follow the same idea.

The price is duplication. A laptop with five Electron apps carries five browser engines and five Node.js runtimes, each in its own folder, each updated by its own vendor, each on its own schedule. When a flaw is found in that shared code, the operating system cannot patch it for you. Every app vendor must ship a new build, and every customer must install it.

FIGURE 5.1

One shared runtime, or one runtime per app



Patch time = slowest supplier + slowest customer install

With a shared runtime that is one vendor. With bundled runtimes it is every vendor on the machine.

Red marks the copy whose vendor has not shipped the fix yet.

Neither model wins. Shared runtimes centralize patching: one fix reaches every app, but how fast depends on that one vendor's cadence, and a change can break apps. Bundled runtimes fix the runtime version but multiply the suppliers you depend on.

HOW IT GOT THIS WAY

The first runtimes were small. A C program carried a startup routine and a library of functions, and the operating system did the rest. POSIX, first published in 1988, tried to make that library the same on every Unix. Then Sun released Java 1.0 on 23 January 1996 with a bold promise: compile once, run on any machine that has a virtual machine. Microsoft answered on 13 February 2002, when it launched the .NET Framework and Visual Studio .NET. The managed runtime brought garbage collection and safer memory to everyday business code. Twenty years later, governments began to ask why so much of the system beneath those runtimes was still written in C and C++.

Memory safety becomes policy

A **memory-safety bug** happens when a program reads or writes memory it should not touch: past the end of a buffer, after the memory was freed, or through a pointer of the wrong type. C and C++ let these mistakes compile. Languages such as Rust, Go, Java, C#, Swift and Python are designed to stop most of them.

Two vendor numbers made this a board-level topic. In July 2019 the Microsoft Security Response Center wrote that about 70 percent of the vulnerabilities Microsoft assigns a CVE each year are memory-safety issues. In September 2024 Google reported that memory-safety issues fell from 76 percent of Android vulnerabilities in 2019 to 24 percent in 2024,

after Android began writing new code in memory-safe languages. Google's point was that most of the old C and C++ code was still there; the change came from writing new code differently.

Governments turned this into guidance. Three documents mark the path:

- **November 2022:** the US National Security Agency published a cybersecurity information sheet, "Software Memory Safety", urging a strategic shift to memory-safe languages.
- **6 December 2023:** CISA, the US cybersecurity agency, with NSA, the FBI and agencies from Australia, Canada, the United Kingdom and New Zealand, published "The Case for Memory Safe Roadmaps", asking software makers to publish their plan.
- **24 June 2025:** CISA and NSA published "Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development".

None of these documents is a law that bans C. They are guidance. But some regulated buyers and their auditors now ask software suppliers for a memory-safety roadmap, and "we never thought about it" is a weak answer.

> *AI lowers the cost of writing and porting code. It does not lower the cost of answering for it.*

Rust, and the cost of moving

Rust is the language most often named in these discussions because it gives memory safety without a garbage collector and without a virtual machine. Its compiler checks who owns each piece of memory, and when it can be freed, before the program runs. That makes it usable in kernels, drivers and other places where a garbage collector would not fit.

Rust still has a runtime, in the plain sense of this chapter. Its standard library carries a module of runtime services that, in the words of its own source, includes "the global heap" and "unwinding and backtrace support", and it sets up the program before your `main` function starts. It is small and it is yours to ship, which means it is also yours to update. Rust code also calls the platform's C library on most systems, so glibc, the UCRT and libSystem stay in the bill.

Moving existing code is the hard part. On 31 July 2024 DARPA, the US defense research agency, announced **TRACTOR** (Translating All C to Rust), a program to automate the translation of legacy C into safe, idiomatic Rust. DARPA did not require AI, though its program manager said large language models will almost certainly be part of the answers.

MIT Lincoln Laboratory leads the testing and publishes benchmarks for the teams. It is a research program. As of November 2026 it shows that the problem is worth funding; it does not show that migration is cheap.

> **EXAMPLE**
>
> A device fleet operator runs a payment kiosk whose card reader library is written in C. An AI assistant can propose a Rust version in days. Someone still has to prove it behaves the same on every reader model, pass the payment certification again, and stay on call when it fails at 2 a.m. in a store.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
`glibc` or the UCRT	GNU project; Microsoft	Distribution vendors; Microsoft	The OS or distribution vendor	The OS release	Yes, through every parser that calls it	Rent
.NET runtime	Microsoft and the .NET community	Microsoft	Microsoft, or you if self-contained	3 years (LTS), 2 years (STS), as of November 2026	Yes, in web and desktop apps	Rent
Node.js inside Electron apps	Node.js and Electron projects	Sponsors and foundation members	Each app vendor that bundles it	The Electron major's end of life (latest three majors supported), then the app vendor's release; not the Node.js LTS clock	Yes, anything the app loads	Rent, and track every copy
Python interpreter in tools	Python core developers	Python Software Foundation and sponsors	Whoever froze it into the tool	About 5 years per version	Often, through file and data parsers	Rent or fund
Your C or C++ core modules	Your team	You	You	As long as you ship them	Depends on what they parse	Build, with a memory-safety roadmap

SECURITY LENS

Who can change it? For the base runtime, the operating system vendor. For bundled runtimes, every app vendor that ships one. A team rarely sees the full list until it scans its machines for copies.

How fast can it be fixed? As fast as the slowest supplier of the slowest copy. A shared runtime is patched once; a bundled one is patched app by app, and the oldest build in the field sets your real exposure.

What does AI change here? AI tools make it cheaper to find bugs in runtimes and to port code to memory-safe languages. They do not change who signs the release, who carries the support clock, or who answers when a translated module fails.

BUSINESS IMPLICATIONS

Security. A regulated company should be able to list every runtime on its machines, its owner and its support end date, and to show a memory-safety roadmap for code it writes in C or C++. Auditors have government guidance to point to; have the answer ready.

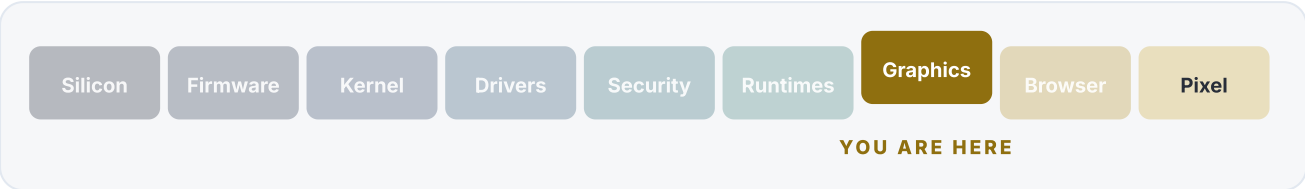
Cost. A software maker that runs Java, Node.js, Python and .NET pays for four support calendars and four upgrade projects a year. Picking one or two runtimes per product line is a cost decision, not a matter of taste.

Value. A device fleet operator gains the most from a short list of shared, vendor-patched runtimes: one update channel to test, one fix to push to every kiosk. Bundle a runtime only where control of the exact version is worth the extra supplier.

This week: write down every runtime you ship, who supplies it and its end date, and treat each bundled copy as a product you now maintain.

CHAPTER SIX

Graphics, Text and the Pixel



Every frame your customer sees is assembled by a compositor you do not own, drawn by a GPU driver whose source you usually cannot read (open stacks such as Mesa and many Linux kernel DRM drivers are the exception), and lettered by a text engine that a handful of people maintain. This layer decides whether your product looks right, whether it can be sold in a given country, and whether a single crafted image or font can take over the machine. It is also the layer most teams never put in a risk register.

Three pipelines, one frame

The prologue followed one key press to the screen. This chapter opens the last stretch of that race: how a finished window becomes light. On all three systems the common composed path has the same shape. The app draws into its own off-screen buffer. A **compositor**, a system program that owns the whole screen, combines every window's buffer into one desktop image. A **graphics driver** hands that image to the GPU and the display hardware, which shows it at the next refresh. That is the common case, not the only one: a full-screen or suitably sized app can bypass composition. Microsoft's DXGI flip-model documentation describes "Direct Flip" and "Independent Flip", where application frames go to the screen without desktop composition, and multi-plane overlays, where the display hardware scans out an app's buffer on its own plane. Linux KMS and macOS have comparable direct-scanout paths.

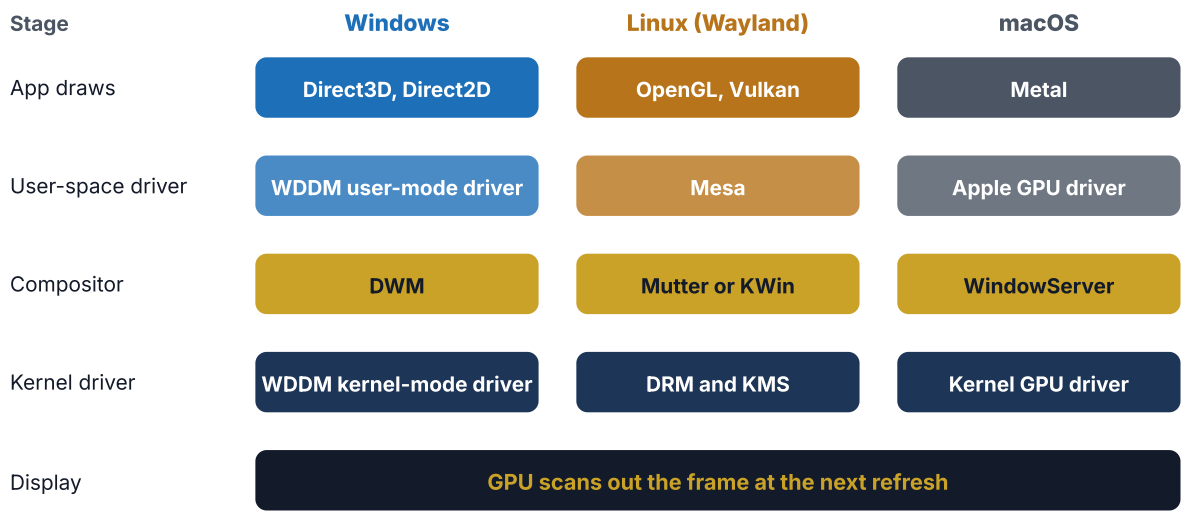
Windows. Apps draw with Direct3D, Direct2D or older interfaces. Microsoft's documentation says that with desktop composition, windows "no longer draw directly to the screen"; their drawing is redirected to off-screen surfaces in video memory, which the Desktop Window Manager (DWM) renders into a desktop image. Composition arrived with Windows Vista, and since Windows 8 DWM is always on and cannot be disabled by users or apps. Underneath sits the **Windows Display Driver Model** (WDDM), also introduced with Vista, which splits each GPU driver into a user-mode part and a kernel-mode part and lets Windows restart a hung display driver without a reboot.

Linux. Apps draw with OpenGL or Vulkan, usually through **Mesa**, the open-source user-space graphics library. The kernel's **Direct Rendering Manager** (DRM) manages GPU memory and jobs, and **Kernel Mode Setting** (KMS) sets the display mode and flips finished frames onto the screen. On a modern desktop the compositor is also the display server: under Wayland, GNOME's Mutter or KDE's KWin receives each app's buffer and builds the frame.

macOS. Apps draw with Metal, Apple's graphics interface announced with iOS 8 on 2 June 2014, or through higher-level frameworks built on it. **Core Animation** describes each window as a tree of layers, and **WindowServer**, the system compositor, combines those layers into the frame the display shows.

FIGURE 6.1

The graphics pipeline on each system



Gold: the compositor, the one program that sees every window. It is always the OS vendor's or the desktop project's.

Same shape, different suppliers. On macOS with Apple silicon, Apple supplies every box except your app. On Windows, Microsoft supplies the graphics interfaces, DWM and the driver model, but the GPU maker writes both halves of the driver, as Chapter Three showed. On Linux the boxes belong to Mesa, the kernel, the desktop project and the GPU maker.

All of it must fit in one frame. At 60 Hz a frame lasts 16.7 milliseconds; at 120 Hz it lasts 8.3. If an app misses that window, the compositor reuses that app's last finished surface, so its window shows a stale frame while the rest of the screen can still update. If the compositor or the GPU misses it, the display can keep showing the previous composed frame, unless a window is on a direct-scanout or overlay path of its own. That is why compositors and drivers are tuned so hard, and why a "small" rendering change from a vendor can cost a kiosk app its smooth scrolling.

Part	Windows	Linux	macOS
Graphics interface	Direct3D 11 and 12	OpenGL, Vulkan (Mesa)	Metal
Compositor	DWM, always on since Windows 8	Mutter, KWin and others	WindowServer with Core Animation
Driver model	WDDM, since Vista	DRM and KMS in the kernel	Apple's own drivers
Text engine	DirectWrite	HarfBuzz shapes, FreeType rasterizes	Core Text shapes and lays out; Core Graphics rasterizes
Who ships fixes	Microsoft and the GPU maker	Kernel, Mesa, desktop project, distribution	Apple

HOW IT GOT THIS WAY

MIT released Version 11 of the X Window System on 15 September 1987. X11 was built for a world of network terminals: a display server drew on behalf of programs that might run on another machine. Decades later most drawing happened in the GPU, and the X server sat in the middle doing little but copying. Wayland took the other path: the compositor is the display server, and apps hand it finished buffers. Wayland 1.0 was announced on 22 October 2012. Fedora 25 made Wayland its default desktop session in November 2016, and GNOME 49 disabled its dedicated X11 session by default in September 2025, while old X11 apps keep running through XWayland.

Text is four problems, not one

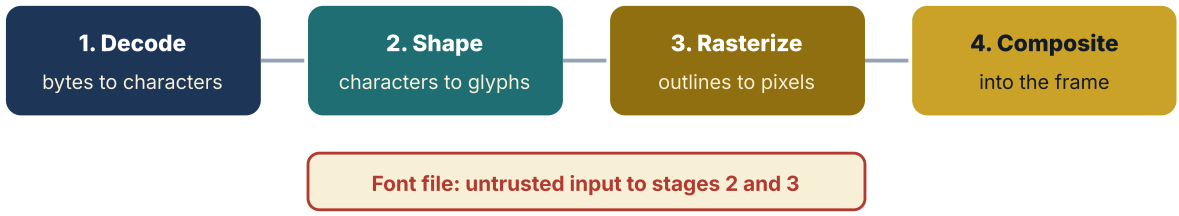
Teams say "fonts" when they mean four different things. Keeping them apart is the first step to managing the risk.

- **Encoding** turns bytes into characters. Unicode, usually stored as UTF-8, is the common answer; national standards still matter.
- **Shaping** turns characters into the right glyphs in the right order and position: joining Arabic letters, reordering Indic vowels, placing marks. HarfBuzz is the open shaping engine used by many browsers and Linux desktops.
- **Fonts** are the files that hold glyph outlines and the rules shaping follows. A font file is a small program of tables, and the engine trusts it to be well formed.

- **Rasterization** turns glyph outlines into pixels at a given size. FreeType does this on Linux and Android. On Windows, DirectWrite does both shaping and rasterizing. On macOS, Core Text shapes and lays out the text and hands glyphs and positions to Core Graphics (Quartz), which draws them.

FIGURE 6.2

The text path: from bytes to lit pixels



Windows: DirectWrite does 2 and 3. Linux: HarfBuzz does 2, FreeType does 3.
macOS: Core Text does 2, Core Graphics (Quartz) does 3.
The compositor in stage 4 is the same one shown in Figure 6.1.

Four stages, often four owners. A web font or an embedded document font reaches stages 2 and 3 straight from the network.

Text is market access

Text decides where you can sell. China's character-set standard **GB 18030-2022** became mandatory on 1 August 2023. It defines three implementation levels, each adding more characters, and it changed how some characters map to Unicode compared with the 2005 edition. Every product in the standard's scope must meet the first level; whether a higher level applies depends on the product's category. A software maker selling into China needs to know which category its product falls in and which level its text stack meets, because a mandatory standard is checked, not assumed.

Myanmar shows the same lesson from the other side. For years most phones in the country used Zawgyi, a font that reused Unicode code points for different letters. The government set 1 October 2019 as the day official electronic communication moved to Unicode. Text typed in one encoding shows as broken letters in the other, so every app had to detect, convert or choose.

Text is also an attack surface. In March 2025 Meta disclosed **CVE-2025-27363**, an out-of-bounds write in FreeType 2.13.0 and earlier, triggered while parsing variable font files, and noted it may have been exploited in the wild. CISA added it to its Known Exploited

Vulnerabilities catalog on 6 May 2025. Newer FreeType releases were not affected. The risk lived in every product that still carried 2.13.0 or an older copy.

> *An SBOM lists the packages you ship. The question that matters is which decoders untrusted bytes can reach.*

Decoder reachability

A **decoder** is code that turns a file format, such as an image, a font or a video, into pixels or glyphs. Decoders parse data that arrives from strangers, so they are where attackers aim.

The lesson of the exploited 2023 flaw in **libwebp**, the WebP image library (Chapters Seven and Eleven), is not that one decoder sits in every path. It does not. The lesson is that a software bill of materials (SBOM), the list of packages in a product, tells you what is installed. It does not tell you which of those decoders untrusted bytes can reach: a chat attachment, a web image, a document thumbnail. That reachability map is what tells you how urgent a fix is. A supplier publishes its answer as a **VEX** statement (Vulnerability Exploitability eXchange), a machine-readable "affected" or "not affected" claim per vulnerability (Chapter 8).

APIs can be withdrawn

Renting a graphics interface means renting its future. In its macOS Mojave 10.14 release notes in 2018, Apple stated that the OpenGL and OpenCL frameworks "are deprecated and remain present for compatibility purposes", and told developers to transition to Metal. Deprecation set a direction, not a removal date. Still, a company whose engineering or medical imaging product was built on OpenGL learned that the future of its rendering layer on the Mac was now set by someone else, even though the code still ran.

SIDEBAR: COLOR IN REGULATED AND ALARM UIS

High dynamic range (HDR) and wide-gamut displays break an old assumption: that a color value always looks the same. ITU-R Recommendation BT.2100, first approved in July 2016, defines HDR television signals with two transfer methods, PQ and HLG. The same red can render differently depending on the display, the OS color management and the HDR mode. For a regulated company with alarm screens or clinical images, test critical colors on the actual fleet hardware, and do not assume the pixel value is the appearance.

SIDEBAR: CUSTOM RENDERERS AND ACCESSIBILITY

Some products draw everything themselves, on an HTML canvas or straight to the GPU, for speed and consistency. Pixels on a canvas carry no meaning for a screen reader, so the product must rebuild its accessibility layer: a parallel description of headings, text and controls. Google Docs began moving from HTML to canvas-based rendering in May 2021 and stated that supported assistive technologies such as screen readers would not be affected. Keeping that promise is ongoing engineering work. The European Accessibility Act applies from 28 June 2025, so owning the renderer means owning that work and its legal exposure.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
System compositor (DWM, Mutter or KWin, WindowServer)	OS vendor or desktop project	OS vendor; distribution vendors	The OS or distribution vendor	The OS release	Indirectly, through every window	Rent
GPU driver (WDDM driver, Mesa and DRM, Apple driver)	GPU maker, Mesa community, Apple	GPU makers and OS vendors	GPU maker or OS vendor	The vendor's driver policy	Yes, through shaders and WebGPU or WebGL content	Rent, and pin tested versions on fleets
Text shaping and rasterizing (HarfBuzz, FreeType)	Small open teams	Mostly sponsors	Every product that ships a copy	Project releases; your copy's age	Yes, web and document fonts	Rent or fund
Image decoders (libwebp, libpng and others)	Open projects, some led by one vendor	That vendor or sponsors	Every product that bundles them	Project releases	Yes, any image a user opens	Rent; map reachability and publish VEX
Your own canvas or GPU renderer	Your team	You	You	As long as you ship it	Yes, whatever it draws	Build only with an accessibility layer

SECURITY LENS

Who can change it? The OS vendor controls the compositor, the GPU maker or OS vendor controls the driver, and every app that bundles a text or image library controls its own copy. The library's maintainers write the fix; they cannot install it for you.

How fast can it be fixed? For system components, as fast as the OS update reaches your machines. For bundled decoders and font engines, as fast as the slowest vendor whose app carries an old copy.

What does AI change here? Image decoders and font engines are small parsers of untrusted files, the kind of code Google's OSS-Fuzz now probes with AI-written fuzz targets (it reported 26 vulnerabilities found that way in November 2024), so expect more advisories against the copies you bundle. Screen-reading agents also turn the rendered pixel into an input to software, so what this layer draws, and who can capture it, matters for security as well as for looks.

BUSINESS IMPLICATIONS

Security. A software maker should map which decoders and font engines untrusted files can reach in each product, and know which copies are bundled. When the next libwebp or FreeType advisory lands, that map turns a week of searching into an hour of patching.

Cost. A device fleet operator should pin and test GPU driver versions for its kiosks and point-of-sale screens, because a driver update that breaks rendering costs field visits. Budget for the OS vendor's API deprecations as planned work, not surprises.

Value. Text support opens markets: GB 18030 level, right-to-left scripts and correct shaping are sales requirements for a regulated company or software maker selling abroad. Owning a custom renderer can buy speed and consistency, but only if you also fund the accessibility work it brings.

Next sprint: for each product, list the decoders untrusted bytes can reach and who patches each one.

CHAPTER SEVEN

The Browser, the Second OS



Open the task manager on any office laptop and count the copies of a browser engine. There is the browser itself, then the chat app, the meeting app, the password manager and the help pane inside your own product. Each of them runs an engine that parses untrusted input from the internet, and each one is patched on someone's schedule. For most business software, the browser engine is now the operating system the code actually talks to.

Three engines, three owners

A **browser engine** is the code that turns HTML, CSS and JavaScript into pixels and behavior. It parses the page, works out the layout, runs scripts, decodes images and fonts, and hands a finished picture to the compositor described in Chapter Six. Three engines matter today, and each has a different owner.

Blink and V8. Blink is the rendering engine of the Chromium open-source project. Google introduced it on 3 April 2013 as "a new open source rendering engine based on WebKit", saying that Chromium's multi-process design had made sharing one engine with other browsers too complex. V8, the JavaScript engine beside it, describes itself as "Google's open source high-performance JavaScript and WebAssembly engine, written in C++." Chromium is open, and companies such as Microsoft contribute to it, but new web features ship through Blink's own "Intent to Ship" process and its API owners. In practice, Google sets the pace.

WebKit. Apple announced Safari on 7 January 2003 with an engine "based on KHTML, from KDE's Konqueror open source project." That engine became WebKit. It is open source under BSD-style and LGPL licenses, and the name and logo are Apple trademarks. Apple sets its direction, and other companies maintain ports, such as WebKitGTK on Linux.

Gecko. Gecko is the engine of Firefox, maintained by Mozilla. It grew out of the Mozilla project, which Netscape started when it opened its browser source code in 1998. It is the only one of the three whose lead owner is not also a platform owner.

On each desktop system, the default browser and the engine that apps can borrow come from different places. The table compares like with like.

Question	Windows	Linux	macOS
Vendor browser and engine	Edge: Chromium (Blink, V8)	No single vendor browser; the distribution chooses	Safari: WebKit
Web view an app can borrow	WebView2 (Chromium)	WebKitGTK (WebKit) or Qt WebEngine (Chromium)	WKWebView (system WebKit)
Who patches the borrowed engine	Microsoft, through the Evergreen runtime; or the app, with Fixed Version	The distribution's package maintainers	Apple, through system and Safari updates
Engine bundled in the app instead	Electron and similar frameworks: one Chromium copy per app, patched by each app maker		
Who sets the engine roadmap	Google, with Microsoft as a large contributor	The upstream project of whichever engine	Apple

How much of the web each engine renders changes every month, so this book does not print a share. The honest summary is that most desktop browsing today runs on Chromium-based browsers, and that WebKit and Gecko remain the two independent alternatives. If a decision depends on the exact share, pull a dated figure on the day you decide.

Microsoft chose to rent the engine

The clearest lesson in this chapter comes from the company that owned the most. Microsoft had its own engine, EdgeHTML, inside the first Edge browser. On 6 December 2018 it announced that it "intend[s] to adopt the Chromium open source project in the development of Microsoft Edge on the desktop." It promised a "Chromium-compatible web platform" and said it would contribute improvements back to Chromium. The new Edge, "rebuilt on the Chromium open source project", left preview on 15 January 2020.

Nobody forced this. Microsoft had the money and the engineers to keep EdgeHTML. It decided that a separate engine was costing web developers testing time and costing its users compatibility, and that the cost of owning was higher than the value. So it moved from **build** to **fund and rent**: it rents Blink's roadmap and pays for a voice in it with its own engineers.

> *Microsoft was not forced out; it chose to stop maintaining its own engine, and bought a voice in the one it rents.*

That is the book's thesis in one move. Owning a layer did not make Edge better. It changed who could fix a web-compatibility defect, and the answer had become "only us, and slowly." After the switch, a fix for a Chromium security bug reaches Edge through the shared upstream project. The security release notes for Edge show this rhythm: Chromium fixes arrive as Edge updates within days.

Embedding: four web views, four update models

Apps rarely build a browser. They borrow one, through a **web view**: a control that shows web content inside a native window. Which web view you choose decides who patches the engine inside your product.

- **WebView2 on Windows.** Generally available for Win32 C and C++ apps on 19 October 2020, with .NET following on 20 November 2020. It comes in two modes. In **Evergreen**, all apps share one copy of the runtime, which "updates automatically without requiring any action from you" and receives the same updates as the Edge browser. In **Fixed Version**, "you control the timing of updates": you package a specific runtime of more than 250 MB with your app, and it is never updated unless you ship a new one.
- **Electron.** Electron describes itself as bundling "the Chromium browser engine and the Node.js runtime into its binary." Every Electron app carries its own engine. The Electron team supports the latest three stable major versions and releases a new major version about every eight weeks. The app maker decides when users get the new engine.
- **WKWebView on macOS and iOS.** Apps use the system's WebKit. Apple patches it with system and Safari updates. The app maker does not ship the engine and cannot patch it.
- **Android WebView.** Android's WebView is Chromium-based. Since Android 5.0, Google's release notes say, "the Chromium layer is now updatable from Google Play," apart from the OS. That applies to devices with Google Play. On Android builds without it, the device maker must ship WebView updates; Chromium's own guide warns that the prebuilt AOSP WebView "is not currently updated on a regular schedule".

None of these is right for every product. A kiosk maker may want Fixed Version so that a browser update cannot change a certified screen. A desktop software maker usually wants Evergreen so that a zero-day is someone else's emergency. The point is to choose on purpose, and to write the choice down.

Update topology is architecture

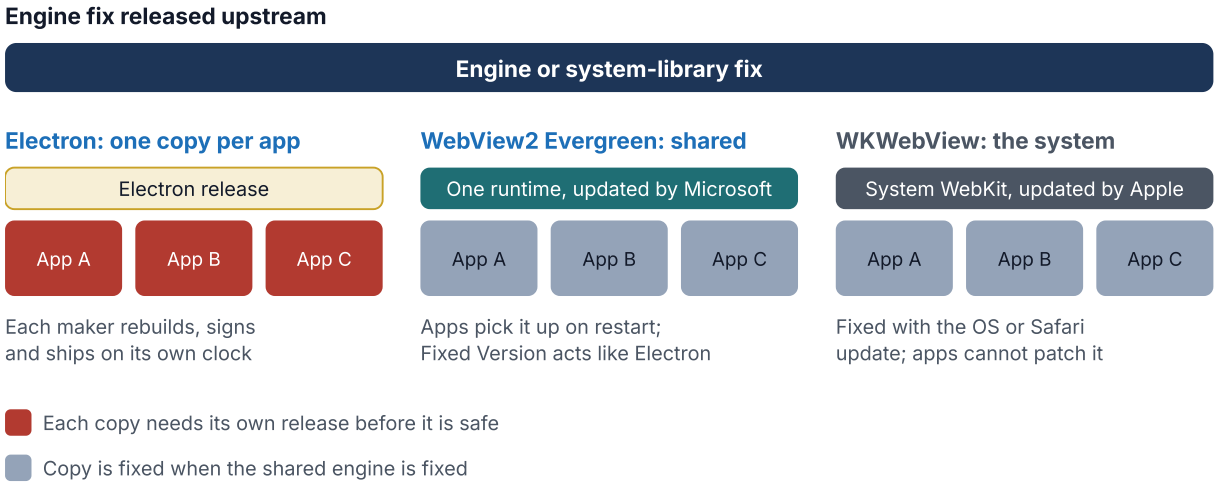
Update topology means the shape of the patch path: how many copies of a component exist on one machine, and who must act before each copy is fixed. It is decided by a dependency choice early in a project, and it shows up years later, in the middle of an incident.

September 2023 made it visible: an exploited flaw in libwebp, the WebP image decoder inside Chromium and many other products (CVE-2023-4863), had to be fixed separately in every bundled copy. Chapter Eleven tells that story in full and measures each owner's delay.

In that incident the topology decided the outcome. An Evergreen WebView2 app got the engine fix with the next runtime update and a restart. Every Electron app had to pull the new Electron release, rebuild, test, sign and ship its own update, on its own schedule, and then wait for users to install it. A Fixed Version WebView2 app was in the same position as Electron. WKWebView apps took a third path. On Apple systems, WebKit decodes images through Apple's ImageIO framework, and Apple's exploited image flaw that week sat in ImageIO (CVE-2023-41064, part of the BLASTPASS attack chain that Citizen Lab reported). Apple has not said which code was at fault, but analyses at the time, such as Ben Hawkes's of 21 September 2023, concluded it was very likely the same WebP bug, since ImageIO had recently added WebP support and Apple had reported the libwebp flaw to Google the day before its own fix. That was Apple's copy of the path, patched once by Apple's OS update for apps that use the system decoders: iOS 16.6.1 and macOS Ventura 13.5.2, on 7 September 2023, fixed it for those apps once the update was installed.

FIGURE 7.1

One engine flaw, three patch paths



Count the red boxes on your fleet: that is the number of owners who must act before you are safe.

Three topologies for one class of flaw. In the 2023 WebP incident, the Electron and WebView2 paths carried libwebp; the Apple path ran through Apple's own system image decoder, ImageIO, very likely with the same bug, and one Apple OS update fixed it for apps that use that decoder. Bundling gives an app control of its engine version and makes it responsible for every patch. Sharing gives the patch clock to the runtime owner.

The browser is built like an OS

A modern browser does the jobs a kernel does, one level up. It runs untrusted programs, separates them, gives them storage, schedules their work and lets them reach the GPU.

Process model. Chromium runs a **browser process** for the user interface, and separate **renderer processes** for web content. Renderers are isolated from each other, so a crash in one usually leaves the others running. They talk to the browser process over an internal message system, the way apps talk to a kernel through system calls.

Sandbox. Because renderers run in their own processes, the operating system can restrict them. In Chromium a renderer reaches the network only through a network service, and the OS limits its access to files, display and input. The browser leans on the sandboxing features of Windows, Linux and macOS; it does not replace them.

Site isolation. Chrome turned on **site isolation** for all sites on desktop in Chrome 67. It aims to keep pages from different sites in different renderer processes, which "helps defend against speculative side channel attacks (e.g., Spectre)." A hardware flaw from Chapter One was answered, in part, by a browser architecture change.

Storage. The browser gives each site its own storage: cookies, local key-value stores and databases, scoped by the site's origin. That is a file system with an access-control rule, run by the browser instead of the kernel.

Compute and graphics. WebAssembly is a compact, low-level code format that browsers run at close to native speed. The W3C announced on 5 December 2019 that the WebAssembly Core Specification had become an official web standard. **WebGPU** gives web pages modern access to the GPU; Chrome 113 enabled it by default on ChromeOS, Windows with Direct3D 12, and macOS. Each new capability is also new attack surface that reaches the GPU driver from a web page.

So a team that ships a web view ships a second operating system inside its product. It inherits that OS's process model, its sandbox, its patch cadence and its bugs.

When the platform owner picks the engine

On iPhone and iPad, Apple's App Review Guideline 2.5.6 says: "Apps that browse the web must use the appropriate WebKit framework and WebKit JavaScript." For years that meant every iOS browser, whatever its brand, was a WebKit browser under the hood. A company could choose Chrome or Firefox on iOS and still be running Apple's engine.

Regulation changed this in two markets. On 25 January 2024, Apple announced changes to comply with the EU Digital Markets Act, including "browser engines, other than WebKit, for browser apps and apps with in-app browsing experiences." The changes reached users in the 27 EU countries in March 2024 with iOS 17.4. As of November 2026, Apple's developer pages offer two entitlements, the Web Browser Engine Entitlement and the Embedded Browser Engine Entitlement: in the EU for devices running "a minimum of iOS 17.4 or iPadOS 18", and in Japan "in iOS 26.2 and later" (Apple's year-based numbering since 2025), where the embedded entitlement is open only to a "browser engine steward", the entity with primary responsibility for a distinct engine. Both pages require a commitment to "mitigate vulnerabilities that are being exploited … in a timely manner (e.g., 30 days for the simplest classes of vulnerabilities being actively exploited)". An app with an embedded engine "must submit an update to your app with that new version within fifteen (15) calendar days" of a new engine version.

For a CTO this is the clearest case in the book of a layer you cannot choose. On iOS outside those markets, the platform owner makes the engine choice. Your web content, your in-app browser and your hybrid app all run on Apple's WebKit, on Apple's patch schedule.

HOW IT GOT THIS WAY

Mosaic, written at the National Center for Supercomputing Applications by Marc Andreessen, Eric Bina and others, was released on 23 January 1993 and made the web visual. Netscape followed, and Microsoft answered with Internet Explorer, bundled with Windows. That fight became the first browser war. On 22 January 1998 Netscape announced it would publish its browser source code, and the Mozilla project grew from it; Firefox 1.0 shipped on 9 November 2004. On 1 September 2008 Google announced Chrome, an open-source browser built on WebKit with its own multi-process design, and launched the beta the next day in more than 100 countries. The later steps, Blink in 2013 and Edge on Chromium in 2020, are told above.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Chromium engine (Blink, V8) in the browser	Chromium project, led by Google	Google, plus contributors such as Microsoft	The browser vendor	Browser release train	Yes, every web page	Rent
WebView2 runtime, Evergreen	Microsoft, on Chromium	Microsoft	Microsoft, automatically	Follows Edge stable	Yes, if the app loads outside content	Rent
Chromium bundled by Electron in your app	Chromium project and Electron team	Google, Electron sponsors, you	You, by shipping a release	Latest three Electron majors	Yes, if the app loads outside content	Rent the engine; build and staff your patch path
System WebKit (WKWebView)	Apple and WebKit contributors	Apple	Apple only	Apple's OS support window	Yes	Rent; required on iOS outside the EU and Japan
Shared image decoder (libwebp)	A small upstream team	Mostly Google	Every product that bundles a copy	Upstream releases	Yes, any image	Rent and fund; track every copy

SECURITY LENS

Who can change it? The engine's upstream maintainer, the vendor of your web view, and, if you bundle, your own release team. On iOS outside the EU and Japan, only Apple can change the engine your web content runs on.

How fast can it be fixed? As fast as the slowest copy on the machine. A shared runtime is fixed when its owner ships and the app restarts. A bundled engine is fixed only when every app maker ships and every user installs.

What does AI change here? Engine code is large, public and full of parsers, and AI already finds flaws in it: Chrome's 28 October 2025 stable release (Chrome 142) credits Google Big Sleep, an AI agent, with four V8 vulnerabilities. Each such find becomes an engine release that every bundled copy must repeat, so the team with the fewest bundled copies has the least work.

BUSINESS IMPLICATIONS

Security. The **software maker** that ships an Electron app controls a browser patch path, whether it planned to or not. Put the Electron upgrade on the same incident clock as your own critical fixes, and measure the days from an upstream Chromium security release to your shipped update.

Cost. Each bundled engine is a copy in memory, on disk and in your test plan. The **regulated company** that must prove what runs on its desktops should count engine copies per machine, and prefer a shared, centrally patched runtime unless it has a written reason to pin.

Value. The **device fleet operator** with check-in kiosks may need Fixed Version to keep a certified screen stable, but then every patch is its job. Pinning trades stability for patch work, so it needs an accountable owner and a review date. Microsoft's own choice shows that renting an engine well can be the better business.

Start here: list every browser engine your product ships or borrows, and name who patches each one and how many days it takes.

CHAPTER EIGHT

Toolchain, Distribution and the Edges of an OS



Every layer in the last seven chapters reached your customer's machine the same way: someone compiled it, packaged it, signed it and pushed it down an update path. That path is a layer too, and it touches all the others. It decides what code a machine trusts, how fast a fix arrives and whether you can recover when a supplier fails. It is also the layer a mid-size company can most realistically control.

Compilers and builds: where every bit starts

A **toolchain** is the set of programs that turn source code into something a machine runs: the compiler, the linker, the build system and the tools that package and sign the result. Each of the three systems has a main native toolchain, and each has a different owner.

Windows. Microsoft's C and C++ compiler, MSVC, ships with Visual Studio. Microsoft writes it, Microsoft pays for it and Microsoft decides when it changes.

Linux. The GNU Compiler Collection, GCC, has built most of the Linux world since GCC 1.0 was released on 23 May 1987. Beside it sits **LLVM**, which describes itself as "a collection of modular and reusable compiler and toolchain technologies." LLVM began as a research project at the University of Illinois. Its C and C++ front end, **Clang**, offers both a GCC-compatible driver and an MSVC-compatible one, so the same compiler family now appears on all three systems.

macOS. Apple ships its own release of Clang, called Apple Clang, inside Xcode, and lists its changes in the Xcode release notes. Apple decides which language features and which warnings reach Apple developers, and when.

The compiler is the first tool to touch every bit in your product. It decides which security hardening is on by default and which language rules are enforced. It also decides whether two builds of the same source produce the same binary.

Packaging and stores: the update path is a product feature

A built program still has to reach a machine and stay current. The **packaging format** and the **distribution channel** decide who signs the code, who can push an update and who can stop one.

Windows. MSIX is the modern Windows packaging format. "All MSIX packages must be signed before installation," Microsoft's documentation says, and on update only the changed 64 KB blocks are downloaded, using the package's block map. **WinGet**, the Windows Package Manager, is an open-source command-line client that installs and upgrades apps from a community repository and other sources. As of November 2026 Microsoft supports it on Windows 10 version 1809 and later, Windows 11 and Windows Server 2025.

Linux. Each distribution runs its own package system: `apt` and `.deb` files on Debian and Ubuntu, `rpm` packages on Fedora and Red Hat Enterprise Linux. Two newer formats cross distributions. **Flatpak** calls itself "a system for building, distributing, and running sandboxed desktop applications on Linux," with shared runtimes underneath. **Snap**, run by Canonical through the Snap Store, checks for updates four times a day by default and confines most apps with kernel features such as AppArmor, seccomp and namespaces.

macOS. Apps reach users through the Mac App Store or directly. **Notarization** is Apple's automated check of a signed app before release. Apple's developer news of 3 September 2019 says that from macOS Catalina, "Mac software distributed outside the Mac App Store must be notarized by Apple." In practice the check is enforced by **Gatekeeper**, which by default expects downloaded software to be signed by a registered developer and notarized. Apple's security guide also says users can override that policy to open other software, unless a device management service restricts it. So, by default and for ordinary users, Apple sits in the release path whether you use its store or not.

Question	Windows	Linux	macOS
Main native compiler	MSVC (Clang available)	GCC and LLVM/Clang	Apple Clang in Xcode
App package formats	MSIX, signed; older MSI and EXE installers	deb, rpm; Flatpak and Snap across distributions	App bundles and installer packages
Who sits in the release path	The Store, if used; signing for MSIX	The distribution's maintainers, or the Flatpak or Snap store	Apple: the App Store, or notarization outside it
Who builds the OS from parts	Microsoft	The distribution, from the kernel community's kernel and thousands of projects	Apple

Look at the update path as a **product capability**, not plumbing. Can you ship a fix to every customer in a day? Can you stop a bad update halfway through a rollout? Can you roll back? A team that can answer yes has built something valuable, even if it rents every other layer in this book. Chapter 11 counts the cost of the opposite case, the 2023 libwebp flaw that forced every app bundling its own browser engine into its own emergency release.

Where an OS ends: the kernel and the distribution

On Windows and macOS the question "who made the operating system?" has one answer. On Linux it has two. The **kernel** is written by a large community, with a new mainline release "every two or three months," according to the kernel's own process guide, and a stable team that backports important fixes. The **distribution**, such as Debian, Fedora, Ubuntu or Red Hat Enterprise Linux, picks a kernel, adds thousands of other projects, patches and builds them, signs the packages and sets the support clock. When you "run Linux," you rent a distribution, and the distribution rents the kernel.

That seam is where a serious supply-chain attack nearly landed. For years `xz`, a compression library used across Linux distributions, was maintained by one unpaid volunteer, Lasse Collin. In 2022 accounts on the project's mailing list pressed him to hand over maintenance; on 8 June 2022 he replied that his "ability to care has been fairly limited mostly due to longterm mental health issues" and that a contributor named Jia Tan might take a bigger role, and by the end of that year Jia Tan was listed as a co-maintainer. On 29 March 2024, an engineer named Andres Freund reported to a public security list that xz had been backdoored. The project's own notice says the 5.6.0 and 5.6.1 release tarballs that contain the backdoor were "created and signed by Jia Tan." Freund noticed it on Debian's unstable branch because of what he called "logins with ssh taking a lot of CPU." Even a test

run of the `sshd` program, not started as a server, took "about 0.5s" on his older system with the backdoored library, against "about 0.01s" without it. The code reached sshd only through a distribution patch that linked OpenSSH to `libsystemd`, which depends on liblzma. Red Hat warned users of Fedora's development branches, and CISA advised rolling back to a clean release such as 5.4.6. According to the first report, the malicious versions had mostly reached distributions only as pre-release packages, so it was caught before the stable releases most companies run.

> *The xz backdoor did not break the kernel or the compiler. It walked in through a small project and a release tarball, and nearly reached the place where a distribution becomes an OS.*

The edge of an OS also moved outward. For a company running services, the **cloud control plane** is now the operating system of the data center: it schedules work, gives out resources and isolates tenants. Kubernetes reached version 1.0 in July 2015, when Google announced it was production ready and donated it to the new Cloud Native Computing Foundation under the Linux Foundation. A Kubernetes cluster has the jobs of an OS, such as scheduling, naming, storage and access control, and it has an owner, a release train and a support clock like every other layer.

Isolation has a unit cost

Every isolation boundary from Chapter Two reappears at data-center scale, and each one has a price per workload. There are four common rungs.

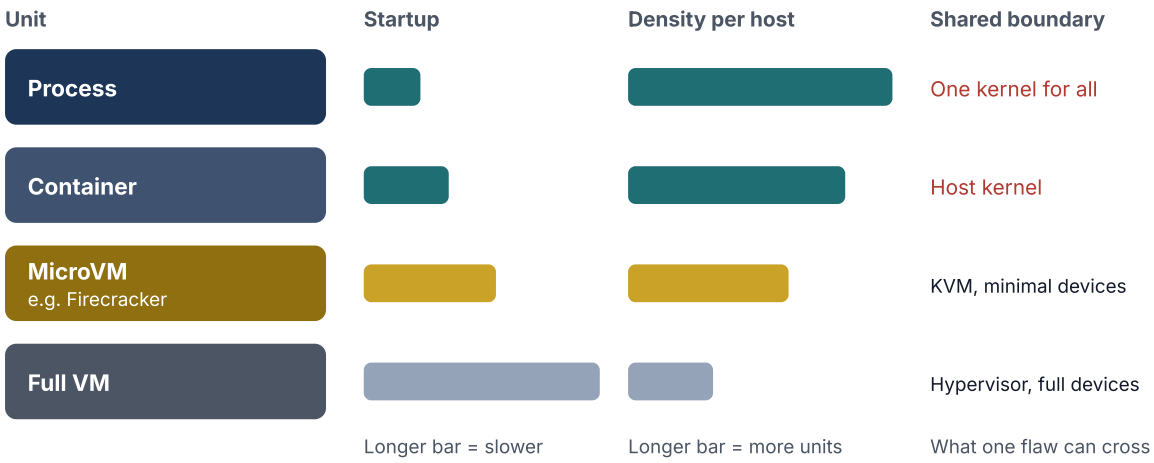
- **Process.** The OS separates memory, but all processes share one kernel. Starting one is cheap and you can run many. A kernel flaw reaches all of them.
- **Container.** A process with its own view of files, network and process IDs, built from kernel features. Still one shared kernel underneath, so the kernel remains the boundary. **gVisor**, written in Go, narrows that boundary with an "application kernel" called the Sentry. The Sentry intercepts the application's system calls and implements them itself; the Sentry in turn uses only a restricted set of host system calls.
- **MicroVM.** A small virtual machine with its own kernel, stripped of devices it does not need. Firecracker runs its guests through KVM, the same hardware virtualization that full virtual machines on Linux use, and offers them only a minimal device model. AWS announced **Firecracker** on 26 November 2018, written in Rust. The announcement said a microVM could launch in "as little as 125 ms" with about 5 MiB of memory overhead,

and that Firecracker already powered AWS Lambda and AWS Fargate. The design is described in the NSDI 2020 paper "Firecracker: Lightweight Virtualization for Serverless Applications."

- **Full virtual machine.** A complete guest OS on a hypervisor that emulates a full set of devices. Slower to start and heavier to run than a microVM. It is not a stronger boundary by default: for both rungs the strength depends on the hypervisor, the device model it exposes and how it is configured.

FIGURE 8.1

The isolation ladder



The two lower rungs give each workload its own kernel, paid for in startup time and density.

Bars are qualitative, not measured. The ladder is not a security ranking between the two virtual-machine rungs: both use hardware virtualization, and a smaller device model means less code an attacker can reach. AWS's own 2018 figures for Firecracker were launch in as little as 125 ms and about 5 MiB of overhead per microVM. Measure your own workloads before you price them.

The choice is a unit cost, not a matter of taste. A software maker running other people's code, such as plug-ins or customer scripts, may pay for microVMs because a shared kernel is too large a blast radius. A team running only its own trusted services may stop at containers and spend the savings on faster patching.

The toolchain is a trust boundary

Reading the source code does not prove what a binary does. A compromised compiler, build server or release script can add behavior the source does not show. Two practices answer this.

Reproducible builds. The reproducible-builds project defines a build as reproducible when, given the same source, build environment and instructions, "any party can recreate bit-by-bit identical copies of all specified artifacts." If two independent parties build the same release and get the same bytes, an attacker who controls only one build machine is caught. It does not help if both parties start from the same malicious source or use the same compromised toolchain: they will reproduce the same bad binary. That is the limit Thompson described in 1984.

The xz case is exactly that limit. Reproducible builds would not have caught it, because the malicious release tarball was the source input: every builder who started from it would have produced the same backdoored library, bit for bit. The trigger for the backdoor "was never included in the Git repository," the xz notice says; it arrived only in the release tarballs, through a modified build macro, `build-to-host.m4`, that the repository did not contain. What helps against that attack is different: comparing each release tarball with the tagged repository and explaining every difference, recording where the source came from (source provenance), and reviewing generated build files, not only hand-written code.

Software bills of materials. An **SBOM** is a formal list of the components inside a piece of software and how they relate. US Executive Order 14028, "Improving the Nation's Cybersecurity," issued on 12 May 2021, directed the Department of Commerce, with NTIA, to publish the minimum elements of an SBOM; NTIA published them on 12 July 2021. An SBOM tells you which packages you ship. It does not by itself tell you that the binary was built from them, which is why it pairs with reproducible builds and signing.

VEX. An SBOM also does not say whether a listed flaw matters. That is the job of **VEX**, Vulnerability Exploitability eXchange: a supplier's machine-readable statement of whether a product is affected by a given vulnerability. CISA's "Minimum Requirements for VEX" (April 2023) allow four statuses: not affected, affected, fixed and under investigation. A "not affected" statement needs a justification, such as "vulnerable code not in execute path"; an "affected" one should say what the customer should do. VEX is the supplier's assertion, not a proof: it states what the supplier believes, and "under investigation" means it does not yet know. Shipping a VEX file beside each SBOM turns "we contain xz" into "we contain xz, and here is whether you need to act."

HOW IT GOT THIS WAY

In his Turing Award lecture, published as "Reflections on Trusting Trust" in Communications of the ACM in August 1984, Ken Thompson described a compiler that would insert a back door into the system's login program. Then he went one step further: the compiler would also recognize when it was compiling itself, and add the same trick to the new compiler. Once the modified compiler existed, the source code could be cleaned and the back door would survive in every future build. No amount of reading the source would reveal it.

Four decades later the lesson still holds. The industry's practical answers each catch a different attack, and none catches all of them. Signing detects a package altered after the signer released it, but not a malicious signer: the xz tarballs were signed by Jia Tan. Reproducible builds detect a tampered build machine, but not bad source. Comparing release tarballs with the repository detects code slipped into a release archive. An SBOM does not detect an attack at all; it tells you where a bad component landed. None of them removes the need to trust the people who write and approve the source.

Recoverability: owning source is not owning recovery

Many teams believe they own a layer because they hold its source code. Ownership is tested on the day a supplier fails: a vendor goes out of business, a signing certificate expires, a maintainer disappears, or a release is found to be malicious. On that day the question is not "do we have the code?" It is whether you can **rebuild** it, **sign** it with a key you control, **deploy** it through your own update path, **roll back** if the fix is wrong and **restore** working machines, and how long each step takes.

So each row of the Pixel Bill of Materials needs one more number: a **tested recovery time**. Tested means a drill that someone actually ran, not an estimate. For rented layers the answer is often "we wait for the vendor," which is honest and acceptable if it is written down and agreed.

EXAMPLE

A hypothetical case. A hotel group runs check-in kiosks on a pinned Linux image with a bundled browser engine. Its drill: assume the engine supplier stops shipping fixes. The team rebuilds the image from source in its own pipeline, signs it with its own key, rolls it to ten kiosks, rolls it back, then rolls it to the fleet. The first drill takes eleven days, mostly because the build needs a machine nobody has kept. After fixing that, the second drill takes two. The number goes into the bill of materials, and the board sees a real recovery time instead of a hope.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Compiler (MSVC, GCC, Clang, Apple Clang)	Microsoft, GNU project, LLVM project, Apple	Vendors and corporate contributors	The vendor; you choose the version	Vendor release cycle	At build time, if it compiles outside code such as dependencies or contributions; not at your customers' runtime. A compromised one taints every build	Rent; pin and record versions
Build and release pipeline, signing keys	Your team, on rented build servers	You	You	Yours	Yes, through dependencies it pulls	Control (build only the trust-critical pieces)
Linux distribution and its packages	Distribution maintainers on upstream projects	Distribution vendor, sponsors, volunteers	The distribution	The distribution's release support window	Yes, through network services	Rent; fund the small projects you depend on
Store or notarization path (Microsoft Store, Snap Store, Apple)	Platform vendor	Platform vendor, from fees or the platform	The platform vendor	The platform's rules	Indirectly	Rent; keep a fallback channel where allowed
Isolation runtime (containers, gVisor, Firecracker)	Open projects, largely led by cloud vendors	Cloud vendors	Your platform team	Project releases	Yes, it holds untrusted workloads	Rent; choose the rung on purpose

SECURITY LENS

Who can change it? Everyone with write access to your source, your dependencies, your build machines, your signing keys and your distribution channel. The xz case showed that a trusted contributor to a small upstream project belongs on that list.

How fast can it be fixed? Only as fast as your update path. A team that can rebuild, sign and ship in hours turns any upstream fix into protection the same day. A team that cannot will wait on every supplier, every time.

What does AI change here? AI coding agents that edit source and build files are new contributors with write access to this layer, so their changes need the same review and signing as a human's. Their commits belong in source-control review, in build provenance and attestations such as SLSA provenance, and in audit logs; the SBOM lists components, not who changed them.

BUSINESS IMPLICATIONS

Security. The **software maker** should treat its build pipeline and signing keys as its most sensitive production system. Pin compiler and dependency versions, build reproducibly where you can, and publish an SBOM and a VEX file with each release.

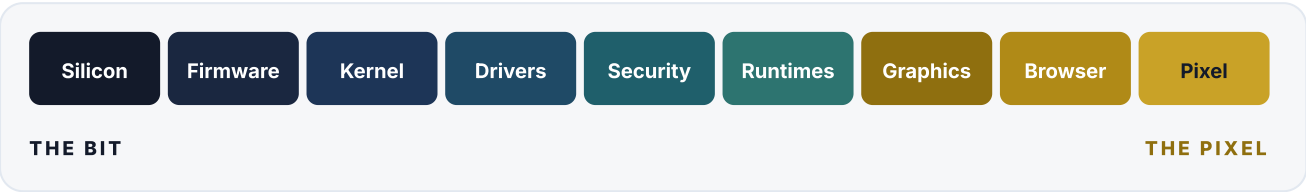
Cost. Isolation is priced per unit. The **regulated company** that runs customer workloads should choose its rung on the ladder on purpose, write down why, and budget for it, rather than inheriting whatever its first cluster used.

Value. The update path is a product capability that customers feel. The **device fleet operator** that can patch every kiosk within a day, and prove it with a drill, has a stronger contract position and fewer emergencies than one with better hardware and no tested recovery time.

This quarter: for every layer you say you own, run one recovery drill and write the tested time in the bill of materials.

CHAPTER NINE

Seventy Years of Operating Systems



Every vendor pitch you hear this year is one swing of a pendulum that has been moving since the 1950s. First one company built the whole machine. Then the machine broke apart into layers, and each layer became a market. Now the largest buyers of computing are pulling the layers back together, down to their own chips. Knowing where the pendulum is tells you which layers you will be offered, and at what price.

The first swing: one vendor, one machine

The first operating systems were written by customers, not vendors. In 1956 General Motors Research and North American Aviation put into production a shared program for the IBM 704, later called GM-NAA I/O. An **operating system** is the program that runs other programs and manages the hardware for them. This one did a simple job: it started the next batch job as soon as the last one finished, and it gave every program the same routines for input and output. That alone raised the number of jobs a 704 could finish each day.

Two ideas followed fast. **Time sharing**, where many users at terminals share one computer in turns too short to notice, was shown at MIT in November 1961 with CTSS, the Compatible Time-Sharing System. And on 7 April 1964 IBM launched System/360, a family of computers of different sizes that could all run the same software. IBM's own history calls it the first time software written for one machine could run on any other in the line. OS/360, the operating system planned for the family, ran late and over budget, and Fred Brooks, who led the work, wrote *The Mythical Man-Month* (1975) about what he learned.

Research went further. Multics, a joint project of MIT, General Electric and Bell Labs, was described in a set of papers at the 1965 Fall Joint Computer Conference. It aimed to be a computer utility, with continuous operation like power and telephone services. IBM's VM/370, announced on 2 August 1972, gave each user a complete **virtual machine**: a software copy of the hardware that could run its own operating system. The cloud still runs on that idea.

In this era the vendor owned every layer, from the circuits to the compiler, and sold them as one product. The first crack came in 1969, when IBM began to price much of its software separately from its hardware. The Computer History Museum puts the effect in one line: overnight, software changed from a giveaway to a competitive commercial product.

The horizontal swing: Unix, C and the PC

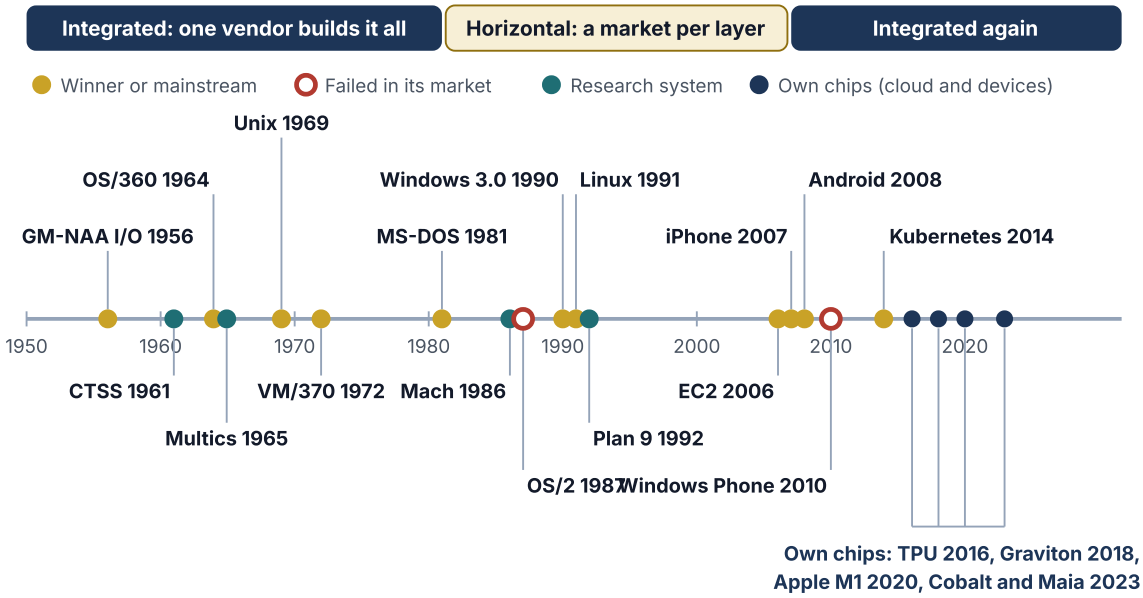
Bell Labs withdrew from Multics in April 1969, and that year Ken Thompson and Dennis Ritchie began Unix, as Chapter 2 told. In the summer of 1973 they rewrote it in C, a language Ritchie designed for the job. Their paper in *Communications of the ACM* in July 1974 made the case in numbers: Unix ran on hardware costing as little as $40,000, and less than two man-years had gone into the main system software. An operating system written in a portable language could move to a new machine with a new compiler, not a rewrite. The OS was no longer tied to one vendor's hardware.

The same break happened on small machines. In 1974 Gary Kildall demonstrated CP/M, the first commercially successful personal computer operating system. Kildall and Glenn Ewing then moved the code that touched the hardware into a separate layer called the **BIOS** (Basic Input Output System), so that CP/M could run on Intel-based computers from many makers. That one design choice made the operating system portable.

IBM launched its Personal Computer on 12 August 1981 with PC DOS, licensed from Microsoft on terms that let Microsoft sell the same system to other makers (Chapter 10 tells how). From then on, the PC was a stack of separate markets: Intel and others sold the processor, Microsoft sold the operating system, dozens of makers sold the box, and thousands of firms sold the software. Windows 1.0 followed on 20 November 1985, Windows 3.0 on 22 May 1990 and Windows 95 on 24 August 1995. Research kept feeding the stack: Mach became part of the base of macOS, and Linux gave the horizontal world a kernel no single firm owned, as Chapter 2 told.

FIGURE 9.1

Seventy years on one line, and the pendulum above it



Dates are first production use, public demonstration, announcement or release, as given in the chapter's sources. The bands are the pendulum: who builds the stack, one firm or a market of firms.

The swing back: integration returns

On 9 January 2007 Apple introduced the iPhone. Apple designed the device and its operating system, and from 2008 it reviewed every app distributed through its App Store. Web apps, in-house apps distributed through Apple's enterprise program and, later, alternative app marketplaces in the EU are the exceptions. On 14 July 2008, days after the App Store opened, Apple reported more than 800 native apps and more than 10 million downloads in its first weekend. The phone was a single product again, the way the mainframe had been.

The same move then reached the desk and the data center. On 22 June 2020 Apple announced that the Mac would move to Apple silicon, and the M1 followed on 10 November 2020, with the unified memory design Chapter 1 described. The cloud providers did the same at their own scale:

- **Amazon Web Services.** Starting with the C5 instance type in 2017, the Nitro System replaced the old virtualization stack with a small hypervisor based on KVM and a set of dedicated Nitro cards. On 26 November 2018, at its re:Invent conference, AWS launched its first instances on Graviton, its own Arm-based server processor.

- **Google.** In May 2016 Google revealed the Tensor Processing Unit, a custom chip for machine learning that was already running in its data centers.
- **Microsoft.** On 15 November 2023, at Ignite, Microsoft announced Azure Maia, an AI accelerator, and Azure Cobalt, an Arm-based CPU. Its hardware leader put the strategy plainly: "we are a systems company."

Why swing back? Because at enough scale the seams between rented layers cost more than owning them. A firm that buys millions of servers can pay for its own chip and recover the cost in power, density and control. A mid-size company cannot. That difference is the core of Part III.

Mobile and cloud: one pendulum, two markets

Mobile split the way the PC did, but faster. Apple kept the integrated model. Google took the horizontal one: on 5 November 2007 it announced Android with 33 other companies in the Open Handset Alliance, and the first Android phone, the T-Mobile G1, was announced in September 2008. Android's user space is licensed mainly under Apache 2.0, a choice Chapter 10 explains. One market, two answers, and both survived.

The cloud went horizontal on top while it went integrated underneath. The Xen hypervisor paper, presented at SOSP in October 2003, showed many operating systems sharing one x86 server with little overhead. Amazon announced EC2, rented virtual machines by the hour, in beta on 24 August 2006. Docker, shown in public at PyCon in March 2013, made the **container** popular: an application packaged with its files and run as an isolated process on a shared kernel, lighter than a full virtual machine. Google open-sourced Kubernetes, a system that schedules containers across many machines, in 2014; its documentation says it combines over 15 years of Google's experience running production workloads. Kubernetes 1.0 shipped on 21 July 2015, and Google gave the project to the new Cloud Native Computing Foundation the same day.

So today you rent a stack that is integrated at the bottom, where one provider controls the chip, the hypervisor and the cards, and more open at the top, where Linux, containers and Kubernetes are available on every major provider. Available is not the same as portable: each provider's managed databases, storage, networking and identity services differ, and those are where a move gets expensive.

> *Linux and Kubernetes run on every major cloud. The managed services around them do not move with them.*

Where each system sits	Windows	Linux	macOS
Born in which swing	Horizontal: an OS licensed to many PC makers	Horizontal: a kernel owned by no single firm	Integrated: one firm, Mach and BSD under its own design
Who makes the hardware	Many PC makers	Anyone; the cloud providers on their own chips	Apple only
Who decides the next layer down	Intel, AMD, Qualcomm and the PC maker	The chip vendor and the distribution	Apple, since its own silicon
Direction today	Pulled toward integration: the Pluton security processor sits inside the CPU and is updated by Windows Update (Chapter 1)	Both: open upstream, integrated inside each cloud	Fully integrated, from chip to App Store

WHAT THE PENDULUM MEANS FOR YOU

A horizontal layer usually offers more suppliers to choose from; an integrated one usually offers a smoother product and less say over the roadmap. Neither is better in general. For each key layer, the useful question is which way it is moving. Workloads on Linux and Kubernetes keep more of a choice of provider than workloads built on one provider's own chips and managed services, though neither move is free.

This layer in the Pixel Bill of Materials

Part II adds no new rows. It adds one question to every row already on the bill: is this layer moving toward one supplier or toward a market, and what happens to your choice if it keeps going?

SECURITY LENS

Who can change it? In integrated eras, one firm can change the whole stack, which makes fixes coherent and makes you dependent on that firm's priorities. In horizontal eras, each layer has its own owner, and a single flaw can need several of them, as Chapter 1 showed with Spectre and Meltdown.

How fast can it be fixed? An integrated vendor can ship one update across chip firmware, kernel and apps. A horizontal stack fixes at the speed of its slowest owner, often a device maker. The cloud gives you both: the provider patches its hosts without you, while you still patch everything above the hypervisor.

What does AI change here? AI assistants lower the cost of reading and porting code from an older era, such as a mainframe batch job or a 32-bit desktop app, so leaving an era is cheaper than it was. Every era you keep running still needs its own patch owner, because old, well documented interfaces are easy reading for attackers' tools as well.

BUSINESS IMPLICATIONS

Security. The regulated company should map each critical system to its era: a mainframe estate, a Windows fleet, Linux in the cloud, phones from one vendor. Each era has a different patch owner and a different evidence trail for auditors, and the gaps sit at the joins between eras.

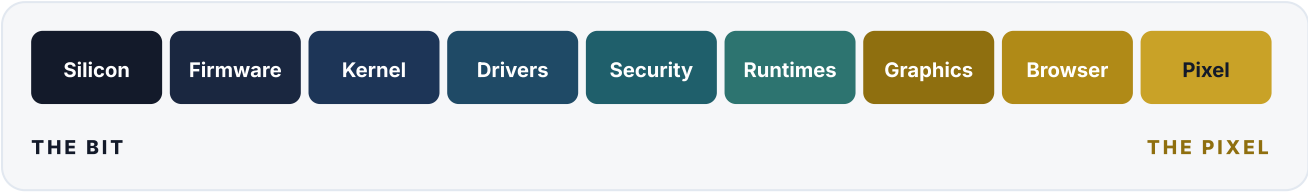
Cost. The device fleet operator buys at the point where the pendulum is horizontal: standard PCs and standard Linux or Windows images, many suppliers, price competition. It should resist moving kiosks onto one vendor's closed stack unless that vendor also takes on the support clock in writing.

Value. The software maker gains by building above the seam. An app that runs on Linux containers and standard browsers can move between clouds and devices as the pendulum swings. Use a provider's own chips where the gain is measured and large, and keep a tested path off them.

First step: for each critical system, write which way its layers are moving, toward one supplier or toward a market, and what you would do if they kept going.

CHAPTER TEN

Why the Winners Won



Engineers like to believe the best system wins. The record says otherwise. Several operating systems that experts admired lost to rivals with more apps, better deals with hardware makers and a promise that old software would keep running. For a CTO this is not trivia: it tells you which layers a company can hope to own, and which layer defeated well-funded firms that tried to own it alone.

Four strong systems that lost

OS/2. IBM and Microsoft announced OS/2 together on 2 April 1987, and version 1.0 shipped in December 1987: a 16-bit, multitasking, protected-mode system with threads and virtual memory, well ahead of DOS. Then Microsoft shipped Windows 3.0 on 22 May 1990. Windows ran on top of the DOS that PC makers already shipped, with the DOS apps users already owned, and Microsoft says Windows 3.0 and 3.1 sold 10 million copies in their first two years. IBM kept OS/2 going for years, then withdrew it from marketing on 23 December 2005 and ended standard support on 31 December 2006.

BeOS. Be Inc., founded by former Apple executive Jean-Louis Gassée, built a fast, multimedia-minded operating system for the 1990s. When Apple looked for a new operating system in 1996, it considered Be and chose NeXT, announced on 20 December 1996. Be then tried to reach PC buyers as a second system next to Windows. In its 2002 antitrust suit, Be alleged that Microsoft's licensing terms kept PC makers from shipping more than one preinstalled operating system, and that a deal with Hitachi fell apart because of them. Be sold its assets to Palm for $11 million in 2001. In September 2003 Microsoft paid Be $23.25 million to settle the suit, without admitting wrongdoing. Whatever the merits, the lesson is about distribution: a system that cannot get onto the machine at the factory has to win each user one at a time.

Plan 9. The Bell Labs team behind Unix began Plan 9 in the late 1980s, to build an operating system for a networked world in which every resource looks like a file. Its first edition, in 1992, went only to universities. The second, in 1995, was sold for $350. It became free over the Internet only in June 2000, and has been under the MIT License since March

2021. By then the market had chosen Unix and Linux. Plan 9's ideas spread anyway. UTF-8, the text encoding of the web, was designed by Ken Thompson and Rob Pike in September 1992 and first put to work in Plan 9. Its file protocol, 9P, is how Windows reaches the files inside a Linux distribution under WSL, since Windows 10 version 1903.

Windows Phone. Microsoft unveiled Windows Phone 7 on 15 February 2010, more than three years after the iPhone was announced. In February 2011 Nokia made it its primary smartphone platform, and on 25 April 2014 Microsoft completed its purchase of Nokia's devices business. On 8 July 2015 Microsoft announced it would record a charge of about $7.6 billion against that business, and Satya Nadella described a move from growing a standalone phone business to growing the Windows ecosystem. Microsoft's current end-of-support table ("What does it mean if Windows isn't supported?") lists Windows 10 Mobile as ending on 14 January 2020; its 2019 Windows 10 Mobile end-of-support FAQ gave 10 December 2019, when most users stopped receiving updates. Microsoft had the money, the developer tools and a large partner. It arrived after the two app stores that mattered were already full.

Four winners, and what they had

Windows won on distribution and compatibility. The non-exclusive DOS license, told in the box below, meant each new PC clone made the Microsoft market larger. Compatibility became a product promise: today WOW64 still runs 32-bit Windows applications on 64-bit Windows, without the user turning anything on. Microsoft's own documentation also shows the limit: 64-bit Windows does not run 16-bit applications. Even the best compatibility promise ends somewhere.

Linux won on servers, supercomputers and embedded devices, and as the kernel inside Android, though not on the consumer desktop. It won on openness and on who paid for it. Linus Torvalds moved the kernel to the GNU General Public License in early 1992, so anyone could ship it, and anyone who distributed a modified version had to provide or offer the complete corresponding source to the people who received it. Changes kept inside one company carry no such duty. On 12 December 2000 IBM said it would spend $1 billion on Linux in 2001. Large companies funding a kernel they did not own is the pattern Chapter 2 described. Since the November 2017 TOP500 list, every one of the world's 500 fastest supercomputers has run Linux.

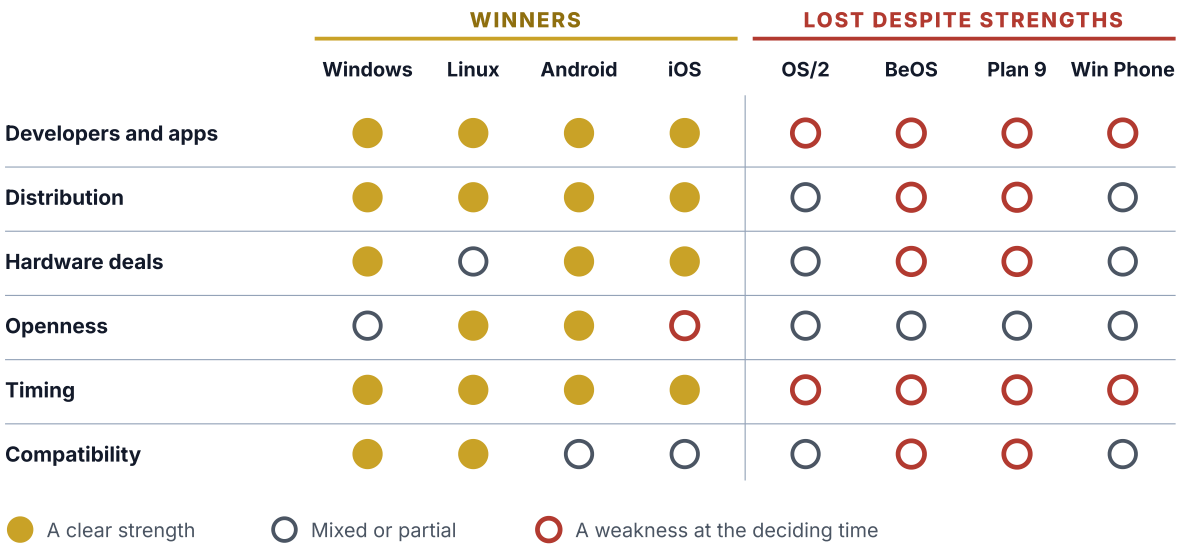
Android won on hardware deals. Google announced it on 5 November 2007 with 33 other companies, including phone makers, chip makers and carriers. Its documentation explains the license choice in business terms: Google prefers Apache 2.0 for Android's user space because device makers ship it as a fixed system image, which suits a permissive license. In May 2021 Google said Android ran on more than 3 billion active devices. The deals also drew

regulators: on 18 July 2018 the European Commission fined Google 4.34 billion euros for tying its apps to Android licensing. In September 2022 the EU General Court largely upheld the decision and set the fine at 4.125 billion euros, and on 2 July 2026 the Court of Justice dismissed Google's final appeal.

iOS won on developers, from inside a closed system. On 6 March 2008 Apple released a free software development kit and set the terms: developers set their own prices and keep 70 percent of sales, and apps are sold only through the App Store, after Apple's approval. In the store's first days, Apple reported more than 800 native apps and more than 10 million downloads. Apple kept the hardware and the store, and left the app layer to everyone else.

FIGURE 10.1

The factors behind each outcome



The author's reading of the cases in this chapter, not a measurement. Each winner was strong on at least four of these six factors. The losers had real strengths the matrix does not show: OS/2 had a more advanced design and IBM's sales channel, Windows Phone had Microsoft's money and tools. iOS won while closed, so openness alone decides nothing.

The lessons, factor by factor

Factor	What it means	Example that shows it
Developers and apps	Users buy a system for what runs on it	Apple's 2008 SDK and 70 percent revenue share; Windows Phone's late store
Distribution	Being on the machine when it is sold	MS-DOS licensed to every PC maker; BeOS shut out of factory installs, by its own account
Hardware deals	Makers and carriers who profit from shipping you	Android launched with 33 partners; the 2018 EU decision shows the same deals carry legal risk
Openness	Others can ship, change and extend it without asking	Linux under the GPL from 1992; Plan 9 free only from 2000
Timing	Arriving before the market settles	Windows 3.0 in 1990 against OS/2; Windows Phone 7 in 2010 against two full app stores
Compatibility	Yesterday's software keeps working	Windows on DOS, then WOW64 for 32-bit apps; the limit is no 16-bit apps on 64-bit Windows

HOW IT GOT THIS WAY

In 1980 IBM, building its Personal Computer, tried to license CP/M from Digital Research. Accounts of why the talks failed differ. The Computer History Museum's version is that they ended in an impasse over financial terms, with IBM wishing to buy outright and Digital Research seeking a royalty on each copy, while Gary Kildall believed the two sides had essentially agreed. What is not disputed is what followed. Microsoft bought an operating system from Seattle Computer Products, licensed it to IBM as PC DOS on a non-exclusive basis, and kept the right to sell it to others as MS-DOS. PC DOS listed at $40, against $240 for CP/M, and most buyers chose the cheaper system. The price and the license, more than the code, shaped the next twenty years of the desktop.

The desktop today, through the same factors

Factor	Windows	Linux	macOS
Who ships it on the machine	PC makers, at the factory	Some PC makers; mostly installed by IT, and the default on cloud servers	Apple, on every Mac
Where the apps come from	Direct downloads, installers and the Microsoft Store	Distribution repositories, Flatpak, Snap and containers	The Mac App Store and direct downloads
Compatibility promise	Strong: decades of Win32 apps, 32-bit through WOW64	Strong for the kernel's user interface; weaker for desktop libraries	Apple moves on when it changes chips or APIs
What it leaves to others	Hardware and most apps	Almost everything: anyone can ship it	The app layer only

The lesson for owners

Look at what each winner owned and what it let others own. Microsoft owned the operating system and let a whole industry build the hardware and the apps. Apple owns the device and the store, and lets other companies own the apps. Google owns Android's direction and its services, and lets many makers build the phones. Linux is owned by no one, and so everyone can build on it. Each winner owned the part that made it different, and cultivated the firms that build on it while it controlled access to the platform.

The losers had to build an ecosystem against a rival that already had one. Be had to persuade users and developers one at a time without the factory install, as its lawsuit, described above, alleged. IBM had to compete with the DOS and Windows base it had helped create. Plan 9 was not freely available until the market had chosen. Microsoft bought Nokia's devices business to keep Windows Phone hardware coming, and wrote it down about fifteen months later. Timing, licensing terms and a rival's conduct all played a part, so no single factor explains any of these outcomes. What the cases share is that an ecosystem is built by thousands of other firms deciding, each for its own reasons, to show up, and in these cases money and good engineering did not buy that decision.

For a mid-size company the rule is direct. Own what makes you different: your product, your data, perhaps one layer where a fast fix is worth real money. Control your release path: rent the build servers, and build only the trust-critical pieces, your pipeline and your keys. Rent the ecosystem layers: the operating system, the app stores, the driver base, the browser engine.

> *OS/2 had the better kernel for its day. Windows had the installed base.*

This layer in the Pixel Bill of Materials

This chapter adds no new rows. It marks three kinds of row on your bill: the ecosystem layers you rent, such as app stores, compatibility layers and device makers' OS images; the row you build, your product; and the row you control, its release path.

SECURITY LENS

Who can change it? The ecosystem owner. A store can remove your app, change its review rules or change what an app may touch, and a compatibility layer can be retired. You rent these layers, so you inherit their owner's decisions.

How fast can it be fixed? An integrated owner can ship a fix to every device it controls. In a horizontal ecosystem the fix waits for each maker, as with Android system images that reach phones through their makers. Compatibility layers are slow to fix because they must keep old behavior, including some old mistakes.

What does AI change here? AI lowers the cost of porting an app to a second platform, so renting two ecosystems instead of one is more affordable than it was. It does not change the store owner's power: review rules and API retirements still arrive on the owner's schedule.

BUSINESS IMPLICATIONS

Security. The regulated company should treat each app store and device maker as a supplier with its own patch clock. Ask for the update window in writing before you buy a phone or tablet model, and retire models when that window ends.

Cost. The device fleet operator should not build its own operating system for kiosks or point-of-sale machines to escape license fees. The history of OS/2, BeOS and Windows Phone says the hidden cost is the drivers, tools and apps no one else will write for you. Rent a mainstream OS and lock it down.

Value. The software maker's value lies in its product, not in the platform under it. Ship on the platforms your customers already use, through their stores and installers, and spend the saved effort on the features that make you different.

Next planning cycle: list every layer you are trying to own; for each one that is an ecosystem layer, write down who already controls it and what it would cost to rent it instead.

CHAPTER ELEVEN

The Seams, in Numbers



Every seam in the stack sends a bill, but few companies add the bills up. A seam is the place where one owner's code hands work to another owner's code. A seam can bring a copy to store, a channel to patch or a delay before the fix arrives, and the costly ones bring all three. Part I named the owners layer by layer. This chapter counts what the seams between them cost, using numbers you can check and a worksheet you can fill in for your own company.

Four things you can count

"Seam tax" is the name this book gives to the cost a company pays because its stack has many owners. It hides inside other budgets: memory on laptops, hours of patch testing, days of exposure after a security fix. It becomes manageable once you split it into four quantities that a team can count:

- **Copies.** How many separate copies of the same runtime or library sit on one machine.
- **Channels.** How many update paths deliver code to that machine, each with its own owner and schedule.
- **Lag.** How many days pass between an upstream fix and the moment your fleet is running it.
- **Reachability.** How many of those copies can be reached by untrusted input, such as a web page, an email or an image.

The three systems shape these numbers differently. The table compares the parts each one gives an app to share, and where copies come from instead.

Question	Windows	Linux (Debian-style distribution)	macOS
Shared web engine an app can borrow	WebView2 Evergreen, patched by Microsoft	WebKitGTK or Qt WebEngine, patched by the distribution	WKWebView, patched by Apple
Where extra engine copies come from	Each Electron app, and each Fixed Version WebView2 app	Each Electron app, and browsers that bundle their own libraries	Each Electron app, and each third-party browser
Update channels the platform offers apps	Microsoft Store, WinGet, or the app's own updater	The distribution's repositories, Flatpak, Snap	Mac App Store, or the app's own updater
Who fixes a flaw in a shared library	Microsoft, for system libraries and the Evergreen WebView2 runtime; each product, for a Fixed Version runtime or another private copy	The distribution, once per source package that carries a copy	Apple for system copies; each vendor for its own

Duplicated runtimes, measured

Electron's own documentation says it embeds "the Chromium browser engine and the Node.js runtime into its binary." Every Electron app therefore carries a whole browser engine, as Chapter 7 explained. The question for a CTO is how big that copy is.

The Electron project publishes its runtime builds on GitHub, with exact file sizes. For Electron 26.2.1, the release that fixed the libwebp flaw on the 26 line, the published archives are 101.8 MB for Windows x64, 97.5 MB for Linux x64 and 90.3 MB for macOS on Apple silicon. Unpacked, the Windows build takes 248.6 MB on disk; the main executable alone is 165.9 MB. These figures are the bare runtime, before the app maker adds a single line of its own code. They describe the version in use during the 2023 incident told below, and sizes change with every release: the Windows x64 archive of Electron 45.3.3, published on 14 November 2026, is 158.3 MB, larger, not smaller. (Archive sizes are from the GitHub release records; the 26.2.1 unpacked size was measured on 17 November 2026.)

WORKSHEET A (AN ILLUSTRATION): ENGINE COPIES ON ONE LAPTOP

Assumptions: a laptop runs five desktop apps built on Electron, such as chat, meetings, a password manager, a note tool and your own product, each on a runtime about the size of Electron 26. This worksheet stands alone; Worksheets B and C below use a different laptop image.

Result: about 5 × 248.6 MB, or roughly 1.2 GB of disk, holds five copies of nearly the same engine. Five copies also mean five patch paths, each on its owner's schedule, and five memory footprints when the apps run at the same time. Memory use depends heavily on each app, so measure it on your own image rather than borrowing a number from a forum.

Disk is cheap. The copies matter because each one is a separate thing to patch, and because each one parses content from outside the company.

Patch lag: one flaw, day by day

This is the full story of CVE-2023-4863, the flaw other chapters point to. **WebP** is an image format, and **libwebp** is the open library that decodes it. On 6 September 2023, Apple's security engineering team and the Citizen Lab reported a bug to Chrome: a heap buffer overflow in WebP handling, a memory error in which a crafted image writes past the end of a buffer. The fix landed upstream the next day, in the code that builds the Huffman tables for lossless images. On 11 September Chrome's stable release shipped it as a critical fix, and Google said an exploit existed in the wild: attackers were already using it.

At first the public record called it a bug "in WebP in Google Chrome." But the flaw was in libwebp itself, and libwebp sits inside other browsers, Electron apps that bundle Chromium, image libraries and server software. Each of those products carried its own copy, and each copy had an owner who had to ship a fix. Here are the events as numbers. Day 0 is 6 September 2023, the report date in Chrome's release note. Dates are publication dates in UTC: Electron tagged all four of its fixed releases on 12 September and published them within about eight hours, either side of midnight UTC.

Day	Date (2023)	Owner	Event
0	6 Sep	Reporters to Chrome	Bug reported
1	7 Sep	libwebp project	Fix committed upstream ("Fix OOB write in BuildHuffmanTable")
5	11 Sep	Google	Chrome stable 116.0.5845.187/.188 ships the fix; exploit known in the wild
6	12 Sep	Microsoft, Mozilla, Electron	Edge 116.0.1938.81; Firefox 117.0.1 and ESR releases; Electron 26.2.1
7	13 Sep	Electron, libwebp, CISA	Electron 22.3.24, 24.8.3 and 25.8.1; libwebp 1.3.2 released; CISA adds the CVE to its exploited list
9	15 Sep	Python imaging library	Pillow 10.0.1 on PyPI
12	18 Sep	One Electron app maker	Visual Studio Code 1.82.2 ships, moving to Electron 25.8.1
15	21 Sep	.NET library makers	SkiaSharp 2.88.6 and Magick.NET 13.3.0 listed on NuGet
?	Later	Your IT team and your users	The last device in your fleet installs every one of these

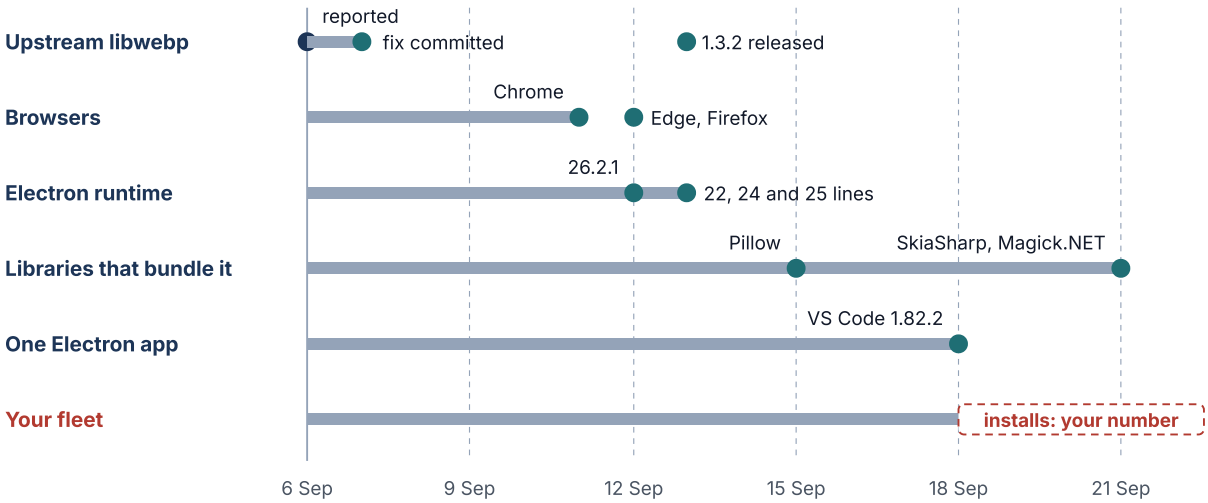
Three facts stand out. First, upstream was first with the fix but not with the release: the fix was committed to libwebp on day 1, Chrome shipped it on day 5, and the official upstream release, libwebp 1.3.2, appeared on day 7. Second, the releases form a branching tree, not one chain. Browser and library makers patched from the upstream fix in parallel, but wherever one product sat on another, it waited: an app built on Electron 25 could not consume an official patched Electron 25 release before Electron 25.8.1 appeared on day 7, unless it backported the fix and built Electron itself, and each such dependency added one to several days. Third, the last row is the one no supplier publishes. It is the row you control, and the one you can measure.

The way an app embedded its engine decided which branch it sat on. An app on the Evergreen WebView2 runtime got the engine fix when Microsoft updated the shared runtime, after a restart. Every Electron app had to pull the new Electron release, rebuild, test, sign and ship its own update, and then wait for users to install it. An app on a Fixed Version WebView2 runtime was in the same position as Electron, because it carries its own

copy. A WKWebView app on macOS sits on Apple's WebKit and Apple's own image decoders, not on Chromium, so it was outside this tree; its engine fixes arrive with Apple's system updates.

FIGURE 11.1

Patch lag across seams: CVE-2023-4863, September 2023



Grey bar: days a copy waited for its owner. Dot: that owner shipped.

The releases branch: rows that bundle another owner's code, such as an Electron app, wait for that owner, while others patch from the upstream fix in parallel. The public record ends at the app maker's release; the time until your last device installs it is yours to measure. Dates from the Chrome, Microsoft, Mozilla, Electron, libwebp, PyPI, NuGet and Visual Studio Code release records.

Update channels on one business PC

An update channel is any path by which new code reaches a machine: the OS updater, a vendor's background service, an app store, or an updater built into an app. Each channel has an owner, a schedule, a test cycle and a way to fail. Nobody publishes a standard count, so here is a worked one.

WORKSHEET B (AN ILLUSTRATION): UPDATE CHANNELS ON A REGULATED COMPANY'S LAPTOP

Assumptions: a Windows 11 laptop in a regulated company, managed by IT, with a typical office toolset. The count is illustrative; your image will differ.

- Windows Update for the OS, plus drivers delivered through it: **1**
- The PC maker's firmware and driver tool: **1**
- Microsoft's updater for Edge and the Evergreen WebView2 runtime: **1**
- A second browser with its own updater: **1**
- The office suite's own update service: **1**
- The Microsoft Store, for the inbox apps it serves: **1**
- Three Electron apps, each with a built-in updater: **3**
- The endpoint security agent, which pulls its own content: **1**
- Your own product's updater: **1**

Result: about 11 channels, each owned by a different team or company. Only one of them, the last, is yours. At least three of them carry a browser engine, and one of them, the security agent, can reach the kernel, as Chapter 2 described.

The count matters for two reasons. Each channel is a supply-chain entry point, so it belongs in the threat model. And each channel can ship a regression on its own day, so when a fleet breaks, the first question is which of eleven owners moved.

The clock got faster, and the queue got longer

Patch lag only matters if attackers move faster than the lag. They do, and the gap has been growing for years. This is the full telling of a figure later chapters cite. Mandiant, part of Google Cloud, measures time-to-exploit (TTE), which it defines as "the average time taken to exploit a vulnerability before or after a patch is released." Its October 2024 analysis of 138 vulnerabilities disclosed in 2023 and exploited in the wild found:

Period	Average time-to-exploit
2018 to 2019	63 days
2020 to early 2021	44 days
2021 to 2022	32 days
2023	5 days (excluding 15 outliers; 47 days with them)

The 2023 average is sensitive to a few extreme cases: Mandiant removed 15 statistical outliers to reach 5 days, and with them the average is 47 days. Read it as a direction, not a constant. The average mixes two kinds of flaw: 97 of the 138, about 70%, were zero-days, exploited before a patch was available, and 41 were n-days, first exploited after one; the 15 outliers removed were 13 zero-days and 2 n-days. Mandiant calls its figures conservative, because it uses the first reported date of exploitation, and real exploitation is almost certainly earlier. The curve was falling well before generative AI tools were widely available, so this data does not show AI causing or speeding it; AI tools are now used on both sides, as Chapter 13 tells. Set the 5-day average against the table above: one widely used Electron app shipped the libwebp fix 12 days after the report, and 7 days after Chrome's public fix. And for a zero-day the attacker's clock starts before day 0, because the exploit exists before anyone reports the flaw: for CVE-2023-4863, Google said an exploit already existed in the wild when Chrome shipped its fix.

The queue of known flaws grew at the same time. One independent count of the public CVE list found about 40,000 records published in 2024 (40,009 at the time of the count) and about 48,000 in 2025 (48,185), each a record year. Exact totals vary between sources, depending on how rejected records are treated. Part of the jump is process, not new danger: the Linux kernel project became its own CVE Numbering Authority in February 2024 and assigns a CVE to any bug fix it judges could matter. The same count credits the kernel with 4,325 CVEs in 2024, about 11% of the year. More records means more triage for every owner in Figure 11.1, whether or not each record is dangerous.

> *Attackers now need days. Your patch path is a chain of owners, and its length is the number you can change.*

Decoder reachability: a number for one shared parser

A decoder is the code that turns a compressed file, such as an image, font or video, into something a program can draw. Decoders parse complex formats from strangers, which makes them a favorite target. No single decoder sits in every pixel path. But one decoder can sit in many paths at once, under many owners, and that is what CVE-2023-4863 exposed.

This book defines **decoder reachability** as a set of counts you can compute for any parser on any machine. They measure different things, so keep them separate:

- **Physical copies (C):** separate copies of the decoder's code on disk. Shared runtimes and hard links make this smaller than the next count.
- **Installations (I):** independently patched installations that contain the decoder. Each one needs its own update.

- **Owners (O):** different owners who must each ship a fix before every installation is safe.
- **Reachable (R):** the installations that untrusted input, such as a web page, a chat attachment or an email preview, can actually reach.
- **Exposure, as two numbers.** The **maximum lag (L)** is the elapsed days from the first public fix until the last reachable installation is fixed; it is the number to set against an attacker's clock. The **copy-days (E)** add up, for each reachable installation, the days it stayed unfixed; they measure exposure volume, the total unfixed exposure the seams create, not the work of patching and not how long an attacker had.

Public records already give partial counts for libwebp in 2023. The GitHub advisory lists 13 affected packages across five ecosystems: npm, PyPI, NuGet, Go and Rust's Cargo. Debian's security tracker records separate fixes in five source packages: `chromium`, `firefox`, `firefox-esr`, `libwebp` and `thunderbird`, because several programs carried their own copy instead of using the system library. Mozilla's single advisory covered five product lines: Firefox, two Firefox ESR lines and two Thunderbird lines.

Counting copies is the first step; the second is knowing which copies can be reached. Ask each supplier for a **VEX** (Vulnerability Exploitability eXchange) statement with its SBOM: an installation its supplier credibly marks "not affected" drops out of R (Chapter 8).

WORKSHEET C (AN ILLUSTRATION WITH STATED ASSUMPTIONS): DECODER REACHABILITY ON ONE LAPTOP

Assumptions: the laptop from Worksheet B, in September 2023. Installations that read web images: Chrome, Edge, the Evergreen WebView2 runtime, three Electron apps and your own product, which embeds Electron. Edge and the WebView2 runtime are separate products with separate installations; Microsoft's documentation says their binaries are hard-linked on eligible systems when both are on the same version, so they may share one physical copy. Exposure is counted from Chrome's fix on 11 September. Chrome and Edge ship on the dates in the table above; the WebView2 runtime is assumed to ship on Edge's day; the four Electron-based apps are assumed to ship 10 days after Chrome; users are assumed to install within 3 days of each release. These delays are assumptions, not measurements.

Result: C = 6 physical copies (7 if Edge and WebView2 are not hard-linked), I = 7 installations, O = 6 owners (Google, Microsoft, three app makers and your own team) and R = 7. The maximum lag L is 13 days: the Electron apps, 10 days to ship plus 3 to install. The copy-days E are 3 (Chrome) + 4 (Edge) + 4 (WebView2) + 4 × 13 (the Electron apps) = 63.

Compare L, not E, with the attacker: 13 elapsed days against Mandiant's 5-day average, and for a zero-day such as this one the attacker started before day 0. E measures exposure volume, not work: seven installations unfixed for one day are seven copy-days, not a seven-day window, and a slow rollout adds copy-days without adding engineering effort. If the four Electron-based apps ran on the shared Evergreen WebView2 runtime instead, I and R would fall from 7 to 3, O for the engine fix from 6 to 2, L from 13 days to 4 and E from 63 to 11, under the same assumptions.

HOW IT GOT THIS WAY

Google announced WebP in September 2010, built on the VP8 video codec, and published libwebp as an open library so that anyone could decode the format. That openness is why it spread into browsers, image libraries and frameworks. The public record lagged too: CVE-2023-4863 was published on 12 September as a Chrome bug, and a separate record filed against libwebp on 25 September was rejected two days later as a duplicate, while every product that bundled the library had to decide for itself whether it was affected.

The seam tax worksheet

Worksheets A to C are illustrations. This blank worksheet turns the chapter into a page a finance committee can read. Fill in one copy per product or device image. Count from an inventory, not from memory, and name one accountable person per line.

Line item	What to count	Count	Accountable owner	Cost driver
Runtimes	Separate copies of each engine or runtime per machine (Chromium, Node.js, JVM, .NET, Python)			Disk, memory, test time per copy
Update channels	Distinct paths that deliver code to the machine			Regressions, supply-chain entry points, change reviews
Patch lag	Days from upstream fix to the last device fixed, per channel, for the last critical CVE			Exposure days, incident hours
Decoder reachability	C, I, O and R, plus maximum lag L and copy-days E, for each image, font and video parser fed by untrusted input			Emergency releases per year
Forks	Components you patch locally instead of using upstream releases			Engineer time to rebase each upstream release
Isolation density	Workloads per host for each isolation unit you use: process, container, VM, microVM			Memory overhead per unit, hosts per workload, cloud bill

Isolation density belongs on the sheet because isolation is a seam too. Chapter 8 gave AWS's own figure of about 5 MiB of memory overhead per Firecracker microVM. The browser pays the same kind of price: the Chromium project reports that full site isolation in Chrome 67 on desktop cost about 10 to 13% more memory with many tabs open. Every boundary that protects you costs memory, and the worksheet makes that trade explicit instead of accidental.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Chromium bundled in each Electron app	Chromium project and Electron team	Google, Electron sponsors, each app maker	Each app maker, by shipping a release	Latest three Electron majors, new major every 8 weeks	Yes, if the app loads outside content	Rent; control the patch path and measure its lag
Shared image decoder copies (libwebp)	A small upstream team	Mostly Google	Every product that bundles a copy	Upstream releases	Yes, any image	Rent and fund; track installations, owners and lag, and ask for VEX
OS update channel	OS vendor or distribution	The vendor	The vendor	OS support dates	Indirectly	Rent
Third-party app updaters	Each software vendor	Each vendor	Each vendor	Each vendor's policy	Yes: each updater downloads and runs code	Rent; count them and cap the number
Your product's release path	Your team	You	You	Your policy	Yes	Control (build only the trust-critical pieces)

SECURITY LENS

Who can change it? At the seams, everyone in Figure 11.1 can change code on your machines: the upstream project, the browser vendors, the runtime team, each library maker and each app maker. You can change only your own release path and how fast your fleet installs.

How fast can it be fixed? As fast as the longest chain of owners for the slowest installation. In the libwebp case Chrome's public fix came on day 5, and fleets could begin installing it that day; bundled copies waited longer, to day 15 for the last libraries in the table and later for each product built on them. Count the hops for your top three runtimes and you know your floor.

What does AI change here? AI bug finders have already reported flaws in widely shared libraries (Chapter 13), and every such finding in a bundled library becomes one fix multiplied by I installations and O owners. AI can help triage the CVE queue, but only design choices, fewer copies and fewer owners, remove a hop from the patch tree.

BUSINESS IMPLICATIONS

Security. The software maker that ships an Electron app sits in the middle of someone else's patch chain. Publish your own lag: the days from an upstream Chromium security release to your shipped update, tracked for every release, and give it the same incident priority as a defect in your own code.

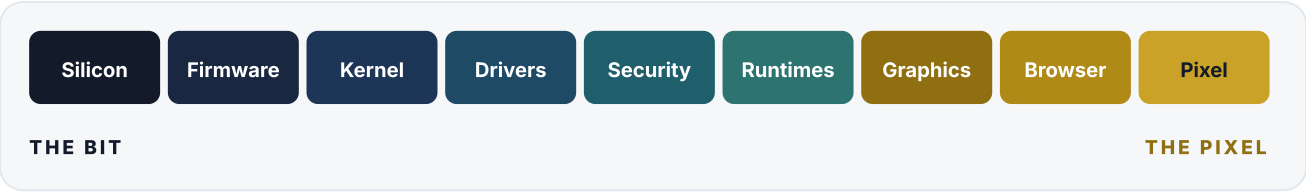
Cost. The regulated company should run the worksheet on its standard laptop image. Every runtime copy and update channel it removes is one fewer item to inventory, test and explain to an auditor, and one fewer owner to chase during the next zero-day.

Value. The device fleet operator pays the seam tax thousands of times over, once per kiosk. Fewer copies and fewer channels on the image mean a shorter last row in Figure 11.1, which is the only row the operator controls and the one its guests feel.

One exercise: pick last year's worst browser-engine CVE and write down, for every installation on your standard image, the day it was fixed on your last device.

CHAPTER TWELVE

The Price of Owning



Owning a layer gives you control, and with it a payroll that runs for as long as the layer does. The code is the cheap part. The expensive parts are the people who understand it, the security response you must now run yourself, the compatibility you must keep, and the clock that keeps ticking after the launch party. This chapter prices those parts, and then shows how real owners fared when they paid them, or stopped.

Five costs that arrive after the code

When a company moves a layer from rent to own, five costs move with it. None of them appears in the build estimate.

- **Hiring.** You need people who can change the layer safely, and enough of them that one resignation is not a crisis. Chapter 8 showed with xz what happens when a critical layer rests on one unpaid maintainer under pressure.
- **Security response.** Someone must read every advisory that touches the layer, judge it, fix it and ship it, on the attacker's clock (Chapter 11 measures it).
- **Compatibility.** Everything above the layer depends on its behavior. Every change you make, and every change you fail to make, breaks someone.
- **Fork upkeep.** If you carry a modified copy of an upstream project, every upstream release is work: merge, rebuild, retest. A fork also carries license duties, covered below.
- **The support clock.** Every layer has an end date, set by you or by your supplier. Devices in the field outlive many of them.

Fork upkeep is the cost teams most often underestimate, because the upstream keeps moving whether you follow or not. Chromium's own process documentation says Chrome ships a new major version to the stable channel every four weeks and refreshes the stable channel weekly. A team that carries its own Chromium changes inherits that cadence.

The license is the other part of a fork's price. The GNU project's FAQ says the GPL does not require you to release a modified version you only use yourself, and that making copies within one organization is not distribution. But once you distribute it, for example on devices you sell or on a product you ship to customers, you must give the people who receive it the complete corresponding source code of that version, not only your changes, under the same license. The Linux kernel is licensed under GPL version 2 only, which lets you either ship that source with the device or include a written offer, valid for at least three years, to supply it. The LGPL, used by many libraries, adds a different duty: version 3, section 4, lets you combine the library with your own code under your terms only if users can relink your app with a modified version of the library, or if you link to it as a shared library they can replace. Budget legal review and a source-release process for every distributed fork of copyleft code.

Large owners show the shape of the choice. Huawei's ArkWeb, the web engine of HarmonyOS, is, in the OpenHarmony documentation's words, "developed based on the Google Chromium kernel." The same page maps each OS release to one Chromium milestone: M99 for OpenHarmony 4.0 and earlier, M114 for 4.1 to 5.1, and M132 for 6.0. Chrome 114 reached stable on 30 May 2023 and Chrome 132 on 14 January 2025. Moving from one base to the next is a project, and between moves the owner carries the gap. That is not a failure. It is the price of owning a fork, paid by a company large enough to pay it.

> **EXAMPLE: THE YEARLY COST OF KEEPING A FORK CURRENT**
>
> **Assumptions:** a software maker carries 40 local patches on top of Chromium inside its product. It follows every major version: 13 a year at a four-week cadence. Each rebase takes 2 engineer-weeks to merge, build and fix conflicts. Each weekly security refresh takes 1 engineer-day to merge and test. Compatibility testing for each major takes another engineer-week. A rotating on-call engineer must be available for emergency releases.
>
> **Result:** 13 × 2 = 26 weeks of rebasing, 52 days (about 10 weeks) of security merges and 13 weeks of testing: about 49 engineer-weeks, or one full-time engineer, before the on-call duty and before anyone builds a feature. Two engineers is the realistic floor, so that one can be on holiday. Change the assumptions to match your patch count; the shape will not change.

Owners that stopped

The history of platforms is full of companies that owned a layer, paid for it for years, and then stopped. The reasons rarely involved bad engineering. Chapter 10 told why some strong systems lost the market; here the question is narrower: what happened to the people who

depended on the layer when its owner stopped paying.

Layer and owner	What happened	Who could fix it afterwards
Copland, Apple's next Mac OS	Shelved in August 1996 in favor of incremental System 7 updates; Apple chose NeXT as its new base in December 1996	Nobody: the code never shipped as a system. Mac OS X, built on NeXT, shipped on 24 March 2001
BlackBerry 10, BlackBerry	BlackBerry said on 28 September 2016 that it would stop developing phones in house; its legacy services for BlackBerry 10 ended on 4 January 2022	Nobody for the service: BlackBerry warned that affected devices "will no longer reliably function, including for data, phone calls, SMS, and 9-1-1"
Windows 10 Mobile, Microsoft	The last version of Microsoft's phone platform; Microsoft's support table lists its end of support as 14 January 2020	Nobody: no more security updates from the owner
Firefox OS, Mozilla	In December 2015 Mozilla stopped selling phones through carriers; in February 2016 it set the end of smartphone development at version 2.6	The community, through forks; the stack moved on to smart TVs
Ubuntu Touch, Canonical	On 5 April 2017 Canonical said it would "end our investment in Unity8, the phone and convergence shell"	The UBports community took over the source and went on developing Ubuntu Touch

The owners were candid about why. Mozilla's announcement said "we could not create a compelling and differentiating end-user value proposition and we failed to build the full ecosystem." Mark Shuttleworth wrote that he had expected convergence on free software to be welcomed by the community and by industry, and "I was wrong on both counts." In both cases the operating system worked. The cost that could not be met was the ecosystem around it: the apps, the carriers, the device makers.

For a CTO, the right-hand column is the lesson. When the owner stopped, the ability to fix the layer either vanished or moved to whoever held the source. Ubuntu Touch survived because its code was open and a community was willing to carry it. BlackBerry 10 devices lost core functions because the service behind them was the owner's to switch off. Before you build on any layer, ask what that column will say if its owner stops.

HOW IT GOT THIS WAY

Copland was meant to be Apple's next Mac OS, planned as Mac OS 8 and built around a new microkernel. In August 1996 Apple shelved it in favor of twice-yearly incremental updates that would carry some of its technology forward. Its microkernel never shipped. On 20 December 1996 Apple announced it would acquire NeXT, and with it the system that became Mac OS X. Apple announced on 9 January 2001 that Mac OS X would ship on 24 March 2001, more than four years after the acquisition. Apple owned the Mac's operating system throughout. Owning did not save Copland; it meant that only Apple could decide to stop, and only Apple could pay for the replacement.

An owned layer that shipped badly, and one its owner chose to stop

Apple Maps, 2012. With iOS 6, Apple replaced the maps in its phones with a service it built itself. Within a week, Apple's chief executive was apologizing. On 28 September 2012 Tim Cook posted a letter to customers: "With the launch of our new Maps last week, we fell short on this commitment. We are extremely sorry for the frustration this has caused our customers." He told users they could try "Bing, MapQuest and Waze" from the App Store, or Google or Nokia maps on the web, while Apple improved its own.

The letter shows the bargain of owning in a single document. Owning the maps layer did not make the product better on launch day. It made it worse. What owning changed was who could fix it: only Apple, on Apple's schedule, with Apple's money. The same letter says "there are already more than 100 million iOS devices using the new Apple Maps," so the fix had to reach all of them through Apple's own update path.

EdgeHTML, 2018 to 2021. Chapter 7 told how Microsoft announced in December 2018 that it would rebuild Edge on Chromium. The closing step came later: Microsoft ended support for the old Edge Legacy desktop app on 9 March 2021, saying it "will no longer receive future security updates." Microsoft could afford to keep building its own engine. It decided that the cost was higher than the value, and moved Edge onto the Chromium open-source project. That was a choice, not a defeat. The next section shows a company that made the opposite choice because it might not be allowed to rent.

Owning because you may not be allowed to rent: HarmonyOS

On 16 May 2019, the US Bureau of Industry and Security added Huawei and 68 of its non-US affiliates to the Entity List, which requires a license for exports to Huawei of items subject to US export rules. Huawei unveiled HarmonyOS at its developer conference on 9 August 2019, describing it as a multi-device operating system built on a microkernel. A layer Huawei had rented from others had become a layer it might not be allowed to rent.

The available figures suggest the bet has held up, mainly in China. Counterpoint Research's quarterly figures, as of November 2026, show HarmonyOS phones at 17% of smartphone sales in China in the third quarter of 2024 and 24% in the second quarter of 2026, ahead of iOS in most quarters shown. Huawei's 2025 annual report says that by the end of 2025 more than 36 million devices ran HarmonyOS 5 or HarmonyOS 6, more than 10 million developers had registered, and AppGallery offered more than 350,000 apps and services. Those are Huawei's own figures.

Two details keep the lesson honest. First, owning made sense here because the alternative was the risk of not being allowed to rent at all. That case does not apply to most mid-size companies: unless export controls, sanctions or a sovereignty mandate can cut off your supplier, HarmonyOS is not a model for your roadmap. Second, even here the owner rents where renting works: as the first section showed, ArkWeb is Chromium-based. Huawei owns the OS and a fork of the engine, and pays to keep that fork moving. Even where owning has held up, it did not remove the price.

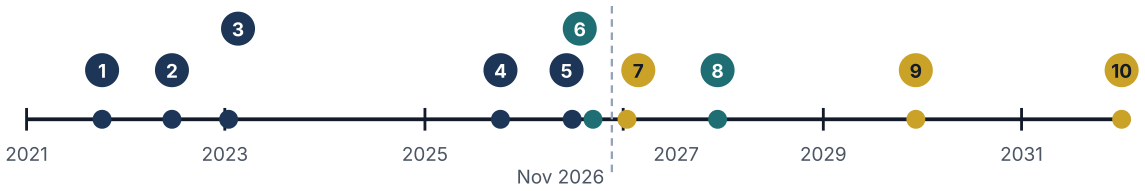
> *Owning puts the fix in your hands. It does not promise that the fix will be good.*

The fleet expiry calendar

The device fleet operator from the Reader's Brief, a hotel group with check-in kiosks or a retailer with point-of-sale terminals, keeps machines for many years. Each machine carries several clocks, and each clock is owned by someone else. The kiosk keeps working when a clock runs out. It simply stops being trustworthy, or stops being fixable.

FIGURE 12.1

Clocks on one kiosk image, 2021 to 2032



1 Sep 2021: Let's Encrypt DST Root CA X3 expires

2 Jun 2022: IE11 desktop app support ends

3 Jan 2023: Chrome 109, last for Windows 7 and 8.1

4 Oct 2025: Windows 10 Home and Pro support ends

5 Jun 2026: Secure Boot 2011 CAs begin to expire

6 Sep 2026: EU CRA vulnerability reporting applies

7 Jan 2027: Enterprise LTSC 2021 support ends

8 Dec 2027: EU Cyber Resilience Act applies in full

9 2029: Edge IE mode, supported "through at least 2029"

10 Jan 2032: Windows 10 IoT Enterprise LTSC 2021 ends

passed · regulation · still ahead

Ten clocks that one kiosk image can carry, each set by a different owner. Dates from Let's Encrypt, Microsoft's lifecycle pages and KB5062710, Google's Chrome support page and the European Commission. The Secure Boot dates are explained in Chapter 1.

Read the calendar from the kiosk's point of view:

- **Certificates.** Let's Encrypt's old root, DST Root CA X3, expired on 30 September 2021. Let's Encrypt warned that devices that did not trust its newer ISRG Root X1 would show certificate errors, and that clients on OpenSSL 1.0.x could fail even if they did. A kiosk with a frozen root store and an old TLS library loses the network services it depends on, on a date nobody at the hotel chose.
- **Browser engines losing support.** Chrome 109, released on 10 January 2023, was the last version for Windows 7 and 8.1. Electron extended its version 22 to 10 October 2023 to cover the same systems. After that, a kiosk on those systems ran an engine that no owner patched.
- **OS support ends.** The same Windows 10 codebase carries very different clocks. Windows 10 Home and Pro reached end of support on 14 October 2025. Windows 10 Enterprise LTSC 2021 reaches end of support on 12 January 2027; Microsoft sells Extended Security Updates for it for up to three more years, to 8 January 2030, at $61 per device for the first year as of November 2026. Windows 10 IoT Enterprise LTSC 2021, the edition built for fixed-purpose devices, is supported to January 2032. Choosing the edition is choosing the clock.
- **Regulation.** The EU Cyber Resilience Act, Regulation (EU) 2024/2847, entered into force on 10 December 2024. Its reporting obligations apply from 11 September 2026 and its main obligations from 11 December 2027. Article 13(8) requires manufacturers to set a

support period that reflects how long a product is expected to be in use, and "the support period shall be at least five years" unless the product is expected to be used for less. For a company that makes or sells connected devices in the EU, the support clock becomes a legal duty.

The expiry calendar is the fleet operator's version of the seam tax. Each clock belongs to a different owner, and the operator controls the one thing that matters: the date by which every device has moved.

Four choices, four prices

Every row of the Pixel Bill of Materials ends in one of four choices. Each has a price, and the price is paid in a different currency.

Choice	What you pay	Who can fix a defect	Typical speed	Main risk
Rent	License or subscription; your team's time to track the vendor's roadmap	The vendor	The vendor's release train	The vendor changes or ends the layer
Fund	Money or engineer time given to an open project's maintainers	The maintainers, with your engineers in the room	Upstream speed, and you hear early	The project still fails if funding is too thin
Fork	The yearly upkeep in the example above, forever, plus license duties when you distribute it	You, for your changes; upstream for the rest	As fast as you choose, after each rebase	You fall behind upstream and own the gap
Build	The whole team, the security response, the ecosystem	Only you	As fast as your own team	You carry every cost in this chapter alone

Funding is the cheapest way to gain influence over a layer, and the most neglected. It is not ownership and does not buy control. It buys a voice, early warning and a healthier project, for a fraction of the cost of a fork. The xz backdoor (Chapter 8) showed what one unfunded maintainer can carry. A company that funds the small projects under its product is paying a premium on an insurance policy whose claim it hopes never to file.

Chapter 15 turns these prices into gates. The short version from this chapter's cases: fork only with a named budget for upkeep, and before you rent, read the right-hand column of the "owners that stopped" table.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Kiosk OS image (Windows 10 IoT Enterprise LTSC 2021)	Microsoft	Microsoft, through your license	Microsoft	To January 2032	Yes, through network and apps	Rent; plan the next image years ahead
A forked browser engine in your product	Chromium project, plus your patches	Google, plus your upkeep budget	You, after each rebase	As long as you fund the rebase	Yes, any page or image	Fork only with a named budget
TLS root store and library on devices	OS vendor or library project	The vendor	The vendor, or you if frozen	Root certificate expiry dates	Yes, every connection	Rent; never freeze it
Small open-source library under your product	A few maintainers	Often nobody	The maintainers	As long as they continue	Often	Fund
Your device support commitment	Your team	You	You	At least five years under the CRA, for products in scope	Not applicable	Build

SECURITY LENS

Who can change it? For a rented layer, the vendor; for a funded one, the maintainers you support; for a fork or a build, you, and only you for your own changes. When an owner stops, the answer changes overnight, as BlackBerry 10 users found in January 2022.

How fast can it be fixed? An owner can fix a layer as fast as its own team can ship, which is fast only if the team exists, is staffed for security response, and has kept its fork current. Apple fixed Maps on Apple's schedule; nobody else could.

What does AI change here? AI assistance makes rebasing, porting and testing a fork cheaper, which shrinks the upkeep example above. It does not staff the on-call security response, meet a license duty or extend a supplier's support date.

BUSINESS IMPLICATIONS

Security. The regulated company should list every layer whose owner could stop, and write down who would be able to patch it the day after. Where the answer is "nobody", that layer needs an exit plan before an auditor asks.

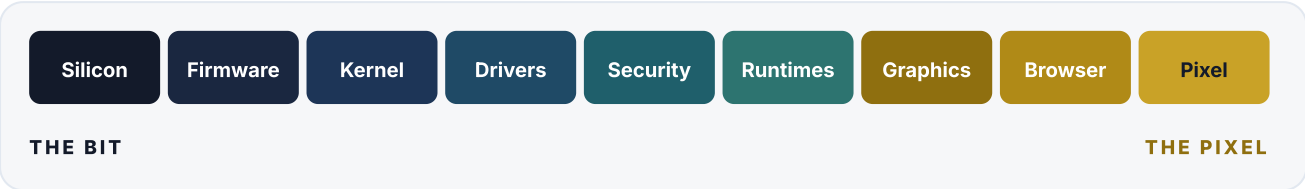
Cost. The software maker that carries its own patches on a large upstream project should put the fork's yearly upkeep on the budget as a named line, with at least two engineers. A fork with no budget is a fork that falls behind.

Value. The device fleet operator should buy hardware and OS editions by their support clocks, not their sticker prices, and keep the fleet expiry calendar next to the depreciation schedule. A kiosk that is cheaper to buy but has a shorter support clock can cost more over its life.

Before the next budget: for each layer you own or fork, name the people who can fix it and the budget that keeps them; for each layer you rent, write the date its clock runs out.

CHAPTER THIRTEEN

AI Moves the Seams



AI did not invent a new kind of security. It changed the prices. Finding a flaw got cheaper, attackers got new tools for a race that was already speeding up, the software you must defend got larger, and a face on a screen stopped being proof of anything. Defenders got the same tools. The one clock that did not speed up on its own is the time it takes your company to ship a fix.

This chapter looks at the stack from the bit to the pixel with that change in mind. It adds a clock to the book's question of who can fix each layer: when flaws are found and exploited faster, the speed of the fix is the part that decides the outcome. Earlier chapters already met pieces of this story. Chapter 4 covered agents that click "I agree" and deepfake video calls. Chapter 5 covered memory safety and AI-assisted translation from C to Rust. Chapter 8 covered the toolchain as a trust boundary. Here the pieces come together.

Four prices that moved

A useful way to read the evidence is as four prices, each with a dated example. None of them is a forecast.

Attacks got cheaper. On 13 November 2025 Anthropic reported that in mid-September it had detected an espionage campaign it attributed, with high confidence, to a Chinese state-sponsored group it designated GTG-1002. The group used Claude Code, Anthropic's coding agent, against about thirty targets and succeeded in a small number of cases. Anthropic's report says the actor used AI "to execute 80-90% of tactical operations independently", with humans at a handful of decision points. The same report adds an important limit: Claude "frequently overstated findings and occasionally fabricated data", claiming credentials that did not work and discoveries that were already public. AI was not a flawless attacker in this case, but it was a cheap one.

Exploits came faster. Mandiant's time-to-exploit average fell for years before generative AI tools were widely available (Chapter 11 gives the numbers and caveats), so that data cannot show that AI caused or accelerated it. What has changed since is that AI tools are

now used on both sides: by attackers, as GTG-1002 shows, and by defenders, as Google's Big Sleep shows in the next section.

The surface got bigger. Every product that adds a language model adds a new kind of input. The OWASP Top 10 for Large Language Model Applications ranked **prompt injection** first in its 2025 edition. Prompt injection means hiding instructions in content the model reads, such as a web page, an email or a document, so that the model follows them. OWASP's 2025 text says it is "unclear if there are fool-proof methods of prevention". The 2026 edition, published in August 2026, keeps prompt injection at number one. The guidance shifted from preventing the trick to limiting what a fooled model can do.

The screen stopped being proof. Chapter 4 told the Arup case: in January 2024 a finance employee in Hong Kong made 15 transfers totalling HK$200 million after a video call in which every other participant was a deepfake. The point for this chapter is narrower. A pixel that shows a person, a document or a confirmation can now be produced cheaply by someone else. Trust has to come from the stack underneath: identity, signing and a second channel.

Defenders got machines too

The same tools work for the defense, and the evidence there is just as concrete.

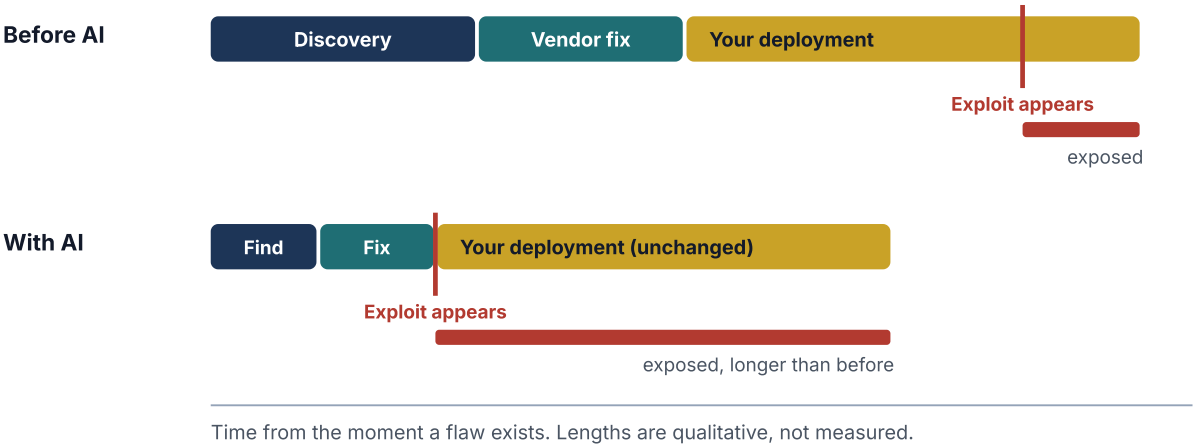
On 1 November 2024 Google's Project Zero and DeepMind described **Big Sleep**, a language-model agent that found an exploitable stack buffer underflow in SQLite, the database engine embedded in browsers, phones and countless applications. The team reported it in early October 2024, SQLite's developers fixed it the same day, and the flaw never reached an official release. In July 2025 Google said Big Sleep had found `CVE-2025-6965`, a SQLite memory-corruption flaw fixed in version 3.50.2. Google's account is that its threat intelligence team saw signs that attackers were staging a zero-day, and Big Sleep helped isolate the flaw before it was used. In August 2025 Google's vice president of security announced Big Sleep's first batch of 20 reported flaws in open-source projects, including FFmpeg and ImageMagick.

The US research agency DARPA ran the **AI Cyber Challenge** (AIxCC) to test this at scale. In the final round at DEF CON in August 2025, according to DARPA, the competing systems "discovered 54 unique synthetic vulnerabilities in the Final Competition's 63 challenges" and patched 43 of them. (DARPA corrected an earlier figure of 70 synthetic vulnerabilities to 63.) They also found 18 real flaws that were not planted, and supplied 11 patches for those. The average time to submit a patch was 45 minutes, at an average cost of about $152 per task. Team Atlanta won the $4 million first prize, Trail of Bits took second and Theori third.

Read these numbers for what they are: a contest on prepared code, and a vendor describing its own tool. They show that machines can find and fix real bugs. They do not show that your fleet is safer. A patch written in 45 minutes protects nobody until it is built, signed, tested and installed, and that work happens on your side of the seam.

FIGURE 13.1

The response clock, before and after AI



Time from the moment a flaw exists. Lengths are qualitative, not measured.

Where AI tools shorten discovery, the vendor's fix and the attacker's exploit, they do not shorten your deployment unless you do. The lengths are an illustration, not a measurement. When the exploit moves left and your deployment does not, the red window grows.

> *AI tools have shown they can find a flaw and write a fix in minutes. Getting that fix onto your machines is still your clock.*

This is the seam argument of Chapter 11 with a clock on it. Exposure is set by the slowest owner on the path between a published fix and a running machine. When a layer has five owners, each with its own release train, the slowest of the five sets your window. Where AI tools make the fast parts faster, the slow parts become more visible. Do not adopt a universal patch deadline from an industry report. Measure your own time from disclosure to mitigation, per layer, and write it down.

The pixel is an input: UI semantics become a versioned integration surface

The screen was not only for people before 2024: scripts written for one application at a time already drove business software through its screens, as the box below tells. What changed in 2024 and 2025 was the arrival of general-purpose agents that use a computer the way a person does. On 22 October 2024 Anthropic released computer use in public beta: a model

that interprets screenshots, moves the cursor, clicks and types. On 23 January 2025 OpenAI released Operator as a research preview, powered by a model it calls the Computer-Using Agent, which "sees" a browser through screenshots and acts through mouse and keyboard.

Screenshots are one way in. The other is accessibility data. Every desktop system exposes a structured description of what is on screen, built so that screen readers can tell a blind user that this is a button labelled "Submit" and that is a text field labelled "Amount". The W3C's Core Accessibility API Mappings specification describes how web content maps to each platform's interface: UI Automation on Windows, ATK and AT-SPI on Linux, and the AX API on macOS. Agent tools now read the same data. Microsoft's Playwright MCP server, for example, lets a model drive a browser "through structured accessibility snapshots", and its documentation says it uses the accessibility tree, "not pixel-based input".

Question	Windows	Linux	macOS
Accessibility interface an agent can read	UI Automation	ATK and AT-SPI	AX API (NSAccessibility)
Who defines the control semantics	Microsoft's UI frameworks, then each app team	Toolkit projects such as GTK and Qt, then each app team	Apple's AppKit and SwiftUI, then each app team
How an app asks to capture the screen	Windows.Graphics.Capture: the user picks in system UI, and a yellow border marks what is captured	On Wayland desktops, the ScreenCast portal of xdg-desktop-portal	The Screen Recording permission under TCC (Chapter 4)

Put those facts together and a consequence follows that few product teams have noticed. Your user interface is now an integration surface. A customer's agent, a partner's automation or your own support bot may depend on the label of a button, the order of fields or the structure of a page. Rename "Submit" to "Continue", move the amount field into a dialog, or replace native controls with a canvas, and someone's automation breaks, silently, in production. Nothing in your API changelog warned them, because the API you changed was the screen.

The answer is to treat UI semantics the way you treat an API. Give important controls stable names and roles. Announce changes to them. Test them with the same accessibility tools a screen reader uses. This has a pleasant side effect: accessibility work and agent readiness are now the same work. A product with a clean accessibility tree serves blind users and gives agents something reliable to read. It is also a necessary part of meeting the European

Accessibility Act that Chapter 6 described, but not all of it: the Act's annex also asks for services that are perceivable, operable, understandable and robust, alternatives to colour and to non-text content, and information on how the service works with assistive technologies. A product that draws everything on a canvas must rebuild that tree, as Chapter 6 explained, and now rebuilds it for two audiences.

HOW IT GOT THIS WAY

Software read other software's screens long before language models. Robotic process automation (RPA) tools drove business applications through their user interfaces when those applications had no API, and the market grew large enough that UiPath, one of its best-known vendors, listed on the New York Stock Exchange on 21 April 2021. Researchers built datasets from screens too: Rico, presented at UIST 2017, collected about 72,000 Android screens from about 9,700 apps, each with a screenshot and the view hierarchy behind it. The habits were already there. Computer-use agents in October 2024 and January 2025 changed the scale and the skill: an agent no longer needed a script written for each screen.

Screen data is AI data

If agents work from screenshots and accessibility snapshots, then those images and trees are data, and they carry whatever was on the screen: customer names, account numbers, medical records, internal prices. Three flows deserve a place in your data inventory.

- **Agent screenshots.** A computer-use agent sends images of the screen to a model to decide its next step. If the model is hosted, those images leave the machine. Where they are stored, for how long, and whether the provider may use them is set by your contract with that provider, not by the agent.
- **OS screen memory.** Microsoft's Recall saves regular snapshots on Copilot+ PCs so a user can search what they have seen. Chapter 4 described its redesign. Microsoft states that the snapshots stay on the device, are encrypted, and are not used to train its AI models.
- **Recordings and replays.** Screen recordings for support, session replay for analytics and screen shares in meetings all create stored images of customer data.

Be precise here, because the debate often is not. **Capture is not training.** A snapshot stored on a laptop is a privacy and security question: who can read it, and what happens when the laptop is stolen. A screenshot sent to a model provider is a data-processing question: what the contract allows. Training is a third question, and it applies only when someone actually builds a dataset, as the Rico researchers did deliberately and openly. Each needs its own control.

This layer in the Pixel Bill of Materials

These are the rows AI adds to the bill. Chapter 15 collects them with every other layer, adds the column **patch time when AI finds it first**, and asks: if a machine found a flaw in this component tomorrow, how long until the fix runs on your machines, and whose clock decides?

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Embedded open-source library (SQLite, FFmpeg, image decoders)	A small maintainer team	Sponsors and volunteers	The maintainers, then every product that bundles it	Project releases	Yes: files and data from outside	Rent and fund; keep the rebuild drilled
Hosted model behind your agent or feature	The model provider	You, per use	The provider only	The provider's model retirement schedule	Yes: prompts, documents and pages it reads	Rent; contract for model-retirement notice, retention of screenshots and prompts, and use for training
Agent identity, permissions and tool list	Your team, on a rented agent framework	You	You	Yours	Yes: every page and file the agent reads	Build the policy: its own identity, least privilege, human approval for actions that cannot be undone
Labels, roles and the accessibility tree of your screens	Your app team, on the OS toolkit	You	You, inside the toolkit's limits	Your release cycle	Indirectly: agents act on what it says	Build; name, announce and test changes as you would for an API

SECURITY LENS

Who can change it? More parties than before. Every party that can ship updates to your machines still can, and now agents acting with your users' privileges can change data and settings too. Content an agent reads, such as a web page or an email, can try to change what the agent does. List agents as actors with write access, the same way Chapter 8 listed contributors and build machines.

How fast can it be fixed? Upstream fixes can be fast: SQLite fixed the first Big Sleep flaw the same day, and AIxCC systems submitted patches in 45 minutes on average on prepared code. Neither example measured deployment, which is the part on your side. Measure the time from disclosure to mitigation for each row of the bill, and work on the longest one first.

What does AI change here? At this layer AI turns the screen, the accessibility tree and every page an agent reads into input, and OWASP's 2025 text says it is unclear whether prompt injection can be prevented reliably; the 2026 edition still ranks it first. The controls that work are old operating-system ideas applied to agents: least privilege, a hard separation between outside content and instructions, and human approval for actions that cannot be undone.

BUSINESS IMPLICATIONS

Security. A **regulated company** should treat every AI agent as a privileged user. Give it its own identity, the narrowest permissions that work, and no authority to approve payments, signatures or access grants alone. Log what it read as well as what it did, so that an injected instruction can be traced after the fact.

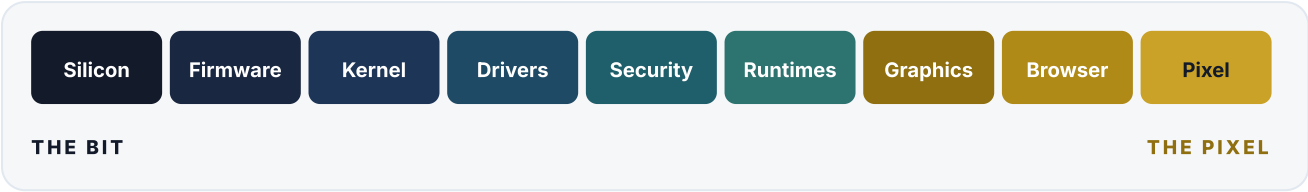
Cost. A **device fleet operator** pays for AI defense twice if it cannot deploy: once for tools that find flaws in the kiosk image, and again for the emergency when a fix sits unshipped. Spend first on the update path, staged rollout and rollback across the fleet, then on detection. Security budgets are growing (Gartner forecast $213 billion of worldwide information security spending for 2025), but most of that money buys tools you rent, and none of it shortens the gap between a published fix and your fleet.

Value. A **software maker** whose screens are stable, well labelled and accessible becomes easier for customers' agents to use, and easier for blind users to use, with the same work. Publishing a change policy for UI semantics, as it already does for its API, turns an unseen liability into a feature customers can rely on.

This week: for each row of your bill of materials, write down the time from a published fix to that fix running on your machines, and shorten the longest one before buying another tool.

CHAPTER FOURTEEN

Controlling the Bit and the Toolchain



A mid-size company will not write its own kernel, and it should not try. But every bit it ships passes through a compiler, a build, a signature and an update channel before it reaches a customer. That path is the layer a small team can most fully control. You still rent the compiler, the app stores, notarization and the OS installers, but the pipeline, the signing keys and the decision to ship can be yours, and together they decide how fast every other fix arrives. If you control one thing from this book, control the path from your source code to your customer's running machine.

Chapter 8 described the toolchain as a layer and a trust boundary, and Chapter 13 argued that, as flaws are found and exploited faster, deployment speed is the part of security that decides the outcome. This chapter turns those findings into a plan. It looks at what the toolchain includes, at what you rent and control in it, at companies that chose to own one layer and rent the rest, and at what AI agents do and do not change about the size of the team you need.

Why the toolchain comes first

The **toolchain**, in the wide sense this chapter uses, is everything between a developer's idea and a running binary on a customer machine:

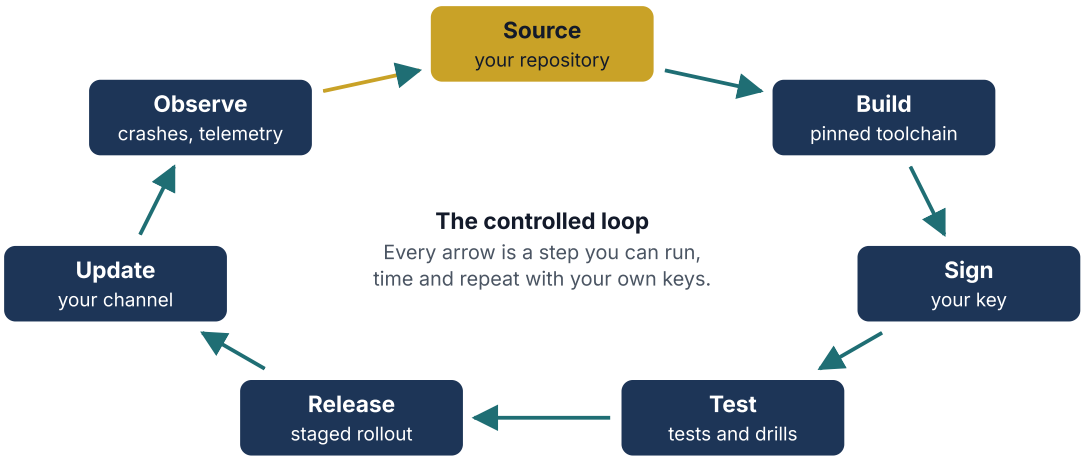
- **the compiler and language version**, which decide what errors are caught before the program runs;
- **the editor or IDE and its assistants**, which now include AI agents that write and change code;
- **the build**, which turns source and dependencies into artifacts, ideally the same bytes every time;
- **packaging and signing**, which decide what a machine will trust;
- **release and update**, which decide how fast a fix reaches every machine, and whether a bad one can be stopped;

- **observation**, the crash reports and telemetry that tell you what to fix next.

Three reasons put this layer first. It is **small enough**: a team of a few engineers can run it, unlike a kernel or a browser engine. It is **leverage**: every fix to a dependency you distribute, such as a bundled runtime, engine or library, reaches your customers only through it, so improving it improves your response for all of them. And it is **largely in your hands by default**, so the real choice is whether you run it well or badly.

FIGURE 14.1

The controlled toolchain loop



Each step has an owner and a time. A team owns the loop when it can run every step itself, with its own keys, and say how long one full turn takes. Observation feeds the next change, so the loop never ends.

The loop has a speed, and the speed is the number to manage. A team that can complete one full turn, from a changed line to a fix running on every machine, in hours, can absorb an upstream security fix the same day. A team whose turn takes weeks is exposed for weeks, whatever tools it buys.

What you rent and what you control

The path has parts you rent and parts you control, and the split looks different on each system. You control a part when you decide how it is composed, signed and released, even if you did not write it.

Question	Windows	Linux	macOS
What you rent (detail in Chapter 8)	The compiler; the Store, if used	The compiler; the distribution or a Flatpak or Snap store	Apple Clang; the App Store or notarization
What stays under your control	The build pipeline, the signing keys, your own update service where the platform allows one, and the decision to ship		

The rented layers are changing their own toolchains too. Chapter 2 gives the status of Rust in the Windows and Linux kernels as of November 2026. For a company that rents the kernel, the lesson is short: driver and agent suppliers will be asked about their language roadmap, and a team that builds its own kernel for a device fleet now carries a Rust compiler in that build.

Owning one layer, renting the rest

The strongest evidence for selective ownership comes from companies that chose one layer, built it, and kept renting everything around it. Each case below is a public, dated engineering decision.

Owned layer	Owner	Language and base	Dated facts	Still rented
Firecracker, a microVM monitor	AWS	Rust, on the Linux KVM hypervisor	Announced 26 November 2018; the post said it powered AWS Lambda and Fargate	The Linux kernel and KVM
Pingora, an HTTP proxy framework	Cloudflare	Rust	Described 14 September 2022; open-sourced 28 February 2024 under Apache 2.0	The OS, the compiler, the hardware
workerd, the Workers runtime	Cloudflare	C++, on V8 isolates	Open-sourced 27 September 2022 under Apache 2.0	The V8 JavaScript engine from the Chromium project
GPUI, the UI framework of the Zed editor	Zed Industries	Rust, drawing with the GPU	Zed open-sourced 24 January 2024; Linux build July 2024; Windows build October 2025	GPU drivers on each platform; the OS text stacks
gVisor, an application kernel for containers	Google	Go	Open-sourced 2 May 2018	The host Linux kernel

Look at why each team built. Cloudflare's Pingora post says that "as Cloudflare has scaled we've outgrown NGINX," and weighs three options: stay with NGINX and possibly fork it, move to a third-party proxy, or build. It warns that moving to another vendor's proxy meant "the same cycle may repeat in a few years". It reports that, under the same traffic, Pingora used about 70 percent less CPU and 67 percent less memory than the old service. The decision was about a layer where Cloudflare's needs had left the rented product behind.

Zed is the pixel-level version of the same choice. Its team built GPUI to render the editor "like a videogame", with shaders on the GPU, Metal on macOS and, in its October 2025 Windows release, DirectX 11 for drawing and DirectWrite for text. Owning the renderer gave Zed control of every frame. It also gave Zed every cost Chapter 6 listed for a custom renderer: its own accessibility work, its own text edge cases, and a new graphics back end for each platform it adds. Notice that even here, Zed rents DirectWrite and Core Text for text. Owning a layer rarely means owning the layers under it.

And look at what each kept renting. Firecracker relies on the Linux kernel's virtualization. gVisor sits on a host kernel. workerd embeds V8. Each team owns one layer it could defend as its difference, and rents the large, shared layers under it.

HOW IT GOT THIS WAY

Rust 1.0 was announced on 15 May 2015. It later became the language of three of the owned layers above: Firecracker in 2018, Pingora in 2022 and Zed's GPUI. gVisor, released the same year as Firecracker, was written in Go, which its announcement said was chosen for its memory and type safety. The pattern is older than either language. Infrastructure companies have long rebuilt one part of their stack when scale made the rented version the bottleneck. What changed in this period is that a safer systems language made such rebuilds less likely to add new memory-safety flaws, which in turn made the decision easier to defend to a security team.

> *Own the layer that makes you different. Rent the large, shared layers under it, and own the path that ships both.*

AI agents and the size of the team

AI coding tools make some of this work cheaper. The honest question is which part, and the evidence points in both directions.

Where the evidence says cheaper. GitHub's controlled experiment, published on 7 September 2022, asked 95 developers to write an HTTP server; those with Copilot finished 55 percent faster on average. Google's DORA research program surveyed nearly 5,000 technology professionals for its September 2025 report and found that 90 percent use AI at work; its summary calls AI's main role "that of an amplifier". On 22 April 2026 Google's chief executive wrote that 75 percent of all new code at Google is now AI-generated and approved by engineers, up from 50 percent the previous autumn. And DARPA's TRACTOR program, described in Chapter 5, funds research on automating translation from C to Rust.

Where the evidence says not so fast. A randomized trial by the research group METR, published on 10 July 2025, followed 16 experienced open-source developers on 246 real tasks in their own projects. With AI tools they took 19 percent longer, although they expected a 24 percent speedup and still believed afterward that they had been faster. In February 2026 METR published an update with a larger group and said its new results were an unreliable signal, likely an underestimate of the benefit, and that the 2025 result should be treated as out of date. That honesty is the point. Measured gains depend heavily on the task, the codebase and the year. The curl project shows both sides in one place: its

maintainer wrote in July 2025 that about 20 percent of that year's security submissions were AI-generated noise, and in September 2025 he praised a set of AI-assisted reports that led to dozens of real fixes.

My assessment, which is a judgment and not a measurement: AI lowers the cost of **writing, porting and reviewing** code, and that makes building a small owned layer more affordable than it was. It does not lower, and may raise, four other costs:

- **Security response.** Someone must triage reports, many of them machine-generated, decide which are real, and ship fixes on a clock Chapter 11 showed is getting shorter.
- **Compatibility.** Every owned layer must keep working on every supported OS version, device and customer configuration. That is testing on real hardware, not writing.
- **On-call.** An owned layer needs people who answer at 2 a.m. and who understand the code, including code an agent wrote.
- **Hiring and continuity.** The team must survive people leaving. Code that only an agent and one engineer understand is a single point of failure.

So the rule for team size is not "AI lets three people own a kernel." It is narrower: AI lets a small team own a small layer, and makes it more important that the team owns the loop that ships it.

Recoverability, layer by layer

Chapter 8 introduced the tested recovery time: the measured time to rebuild, sign, deploy, roll back and restore a layer after a supplier fails. For the toolchain it becomes a set of drills, each with a number. Can you rebuild last quarter's release from a clean machine and get the same bytes? Can you sign with a key held in hardware you control, and revoke it? Can you stop a rollout halfway and roll back the machines that already took it? Can you install a new compiler version and ship within a day if the old one has a flaw?

Run each drill once a quarter and record the time. For a rented layer, the honest number may be "until the vendor ships, plus our deployment time"; write that down too. The goal is not that every number is small. It is that every number is known, so that a board, an auditor or a customer gets a measured answer instead of a hope.

EXAMPLE

A hypothetical regulated company ships a desktop client to its branches. It rents the OS, the compiler and the UI framework, and owns the build, signing and update service. Its first drill finds that a clean rebuild takes three days, because two dependencies are fetched from a public mirror at build time and one signing step needs a specific person's laptop. It vendors the dependencies, moves the signing key into a hardware security module with two approvers, and the second drill takes four hours. Nothing about the product changed. Its exposure to every upstream flaw just fell by days.

This layer in the Pixel Bill of Materials

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Compiler and language toolchain (MSVC, GCC, Clang, rustc)	Vendors and open projects	Vendors, sponsors, corporate contributors	The project; you choose the version	Project release schedule	At build time, if it compiles outside code such as dependencies or contributions; a compromised one taints every build	Rent; pin versions and practise upgrading in a day
AI coding agents in the editor and pipeline	The tool vendor and the model provider	You, by subscription	The vendor	The vendor's terms	Yes: they read issues, pages and files	Rent the tool; own the review and permissions
Build system and dependency mirror	Your team, on open tools	You	You	Yours	Yes, through what it fetches	Control (build only the trust-critical pieces); make builds reproducible
Signing keys and update service	Your team	You	You	Yours	Yes: the update channel faces every machine	Control (build only the trust-critical pieces); hardware-held keys
One differentiating layer (a proxy, runtime or renderer)	Your team, perhaps on open source	You	You	As long as you ship it	Often yes, by design	Build or fork, only with a tested recovery time

SECURITY LENS

Who can change it? Everyone with write access to your source, your build machines, your dependency sources, your signing keys and your update service. AI agents that open pull requests or edit build files belong on that list, with their own identities and no ability to merge or sign alone.

How fast can it be fixed? As fast as one full turn of the loop in Figure 14.1, measured in a drill. Most of that answer is yours; store review, notarization and the compilers you rent add their own time. Shortening the part you control improves your response for every rented layer at once.

What does AI change here? It makes writing and porting code cheaper, which makes a small owned layer more affordable, and it adds new contributors with write access plus a larger stream of bug reports to triage. Review, reproducible builds, provenance and signing apply to an agent's change exactly as to a person's.

BUSINESS IMPLICATIONS

Security. A **software maker** should treat its build and signing system as production, with access reviews, hardware-held keys and two-person release approval. It is the one system through which an attacker can reach every customer at once.

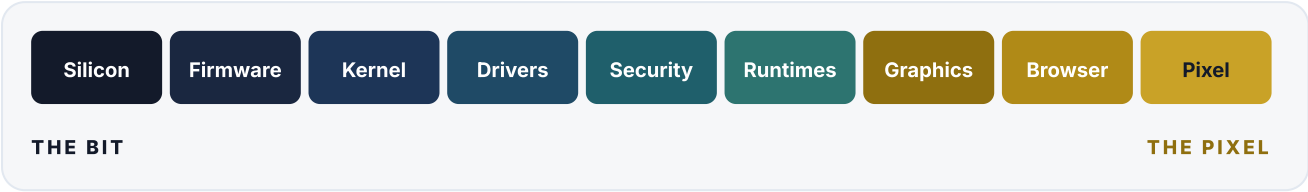
Cost. A **device fleet operator** gets more from an update service with staged rollout and rollback than from owning any other layer. It costs a small team and pays back the first time a fix must reach every kiosk in a day. Budget the people, not only the tools: owning the loop means on-call.

Value. A **regulated company** that can show a reproducible build, a signed release and a tested recovery time for each layer answers audit questions with evidence. That shortens audits and sales cycles, and it turns "what do you own?" from a vague question into a list with numbers.

Each quarter: time one full turn of your toolchain loop, from a one-line change to that change running on every machine, and make it shorter than the last.

CHAPTER FIFTEEN

What to Own, What to Rent



Fourteen chapters have named the owners between your code and your customer's screen. Now you have to decide which of those owners should be you. This chapter turns the book into a method: nine questions for each layer, four honest choices, one consolidated Pixel Bill of Materials, and a plan you can start on Monday. The method will not tell you to own more. It will tell you where owning a layer speeds up the fix enough to be worth the cost.

Nine questions for every layer

Most own-or-rent debates start with a technology preference and end with a budget fight. Start instead with the same nine questions for every row of the bill. Each one has a plain answer, and each answer rules some choices out.

Question	What it asks	What a weak answer rules out
1. Differentiation	Would a customer notice, or pay more, if this layer were better than your competitors' copy of it?	If no, owning it needs another reason: continuity, sovereignty or cost, or no rentable option at all.
2. Risk	What happens to the business if this layer fails, is compromised or disappears?	If the answer is "little", spend nothing beyond renting it well.
3. Support clock	When do updates stop, or a key or certificate expire?	If the clock ends before your product does, renting needs an exit plan.
4. Reachability	Can a file, page, packet, image or font from outside reach this code?	If yes, owning it means owning a stream of security fixes.
5. Patch time	When a flaw is found, possibly by an AI tool before the maintainer, how long until a fix runs on your machines?	Owning is acceptable only with a residual risk you have explicitly assessed and accept. If security is the reason to own, it must leave less risk than renting: you patch as reliably, strip features or privilege you do not need, or keep patching after the supplier stops.
6. Tested recovery time	If this layer breaks on every machine at once, how long until you are back, measured in a drill, not estimated?	If you have never measured it, you do not know whether you can own it.
7. Talent	Can you hire and keep at least two people who understand this layer deeply?	One person is a single point of failure, not a capability.
8. Exit cost	What would it cost, in money and months, to move to another owner?	A high exit cost on a rented layer is a risk to manage, not ignore.
9. Accountable owner	Which named person in your company answers for this layer when it fails?	If nobody, no choice you make is real yet.

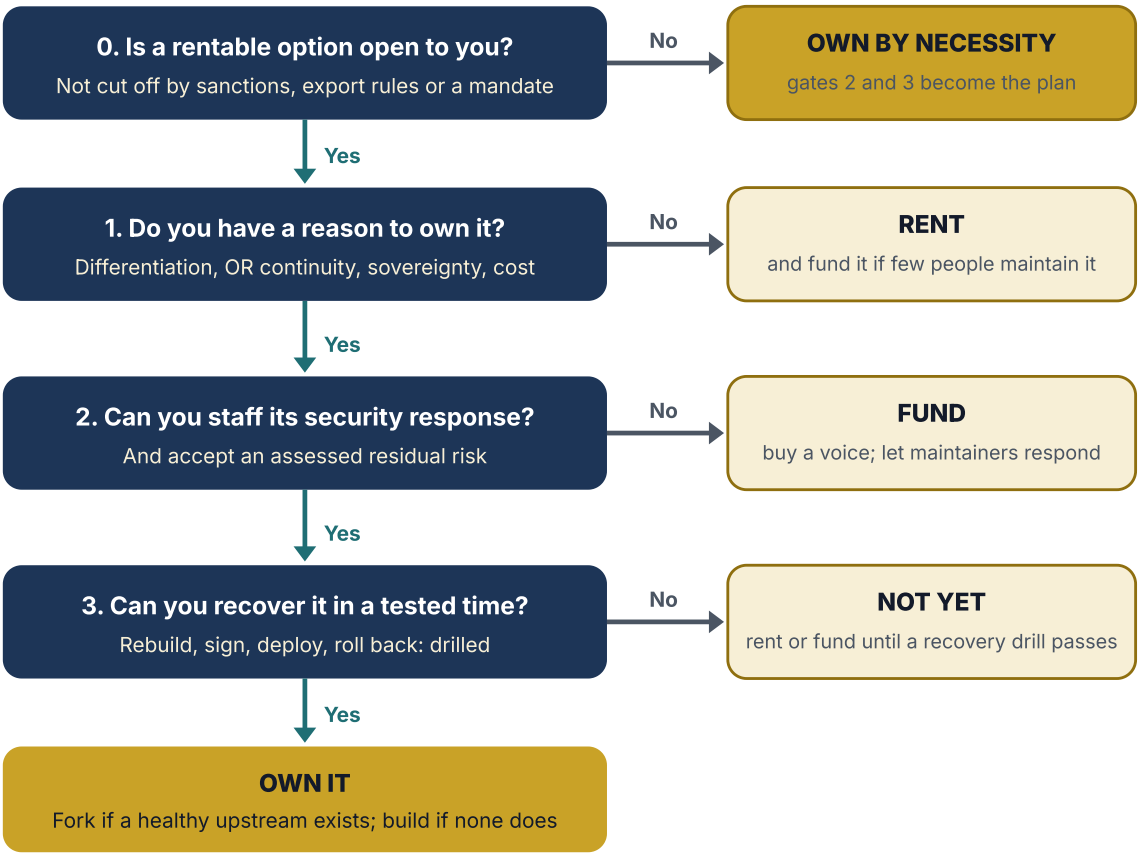
The questions fall into three groups. The first two ask whether you have a reason to own the layer. Questions three to six ask whether you can respond when it breaks. The last three ask whether the company can carry the decision for years. The first gate is an OR: owning can be justified by differentiation, **or** by continuity, sovereignty or cost, for example when a supplier's support ends before your product does, when you may not be allowed to rent, or when renting costs more over the product's life. The next two gates are not optional: whatever the reason, own a layer only when you can staff its security response and its recovery.

The security gate is not a race against the current owner's patch time. It asks for an explicit assessment of the risk left over once you own the layer, and a decision that this residual risk is acceptable. When the reason to own is differentiation, continuity, sovereignty or cost, acceptable is enough. When the reason is security itself, the residual risk must be lower than when you rent. Owning can lower it by patching faster, but also by removing features you do not use, by running the layer with less privilege, or by patching it after its supplier stops.

One case comes before the gates. When no rentable option is open to you, because export controls, sanctions or a sovereignty mandate cut off the supplier, owning is not a choice. Chapter 12 showed that case for HarmonyOS. The other two gates then stop being a test and become the plan: staff the security response and drill the recovery before you depend on the layer. Most mid-size companies never face this branch.

FIGURE 15.1

The decision flow for one layer



A necessity check, then three gates, in order. If no rentable option is open to you, owning is forced and the later gates become the staffing plan. Otherwise the first gate passes on any one reason: differentiation, or continuity, sovereignty or cost. Most layers leave there, and that is the right answer for them. A layer reaches "own it" only when it also passes the other two: its residual security risk is explicitly assessed and acceptable (and lower than renting, if security is the reason to own), and you have proved you can recover it.

Four choices, four rules

Rent means you use the layer as its owner ships it and accept the owner's roadmap. The rule: **rent when you have no reason of differentiation, continuity, sovereignty or cost to own the layer, or when owning it would leave a residual risk you cannot accept.** That covers most of the stack for most companies: the kernel, the compositor, the GPU driver, the system web view. Renting well is a skill. It means pinning versions on fleets, staging updates, tracking support dates and writing exit clauses into contracts.

Fund means you pay the people who maintain an open layer, directly, through a foundation or through a vendor that employs them. The rule: **fund when you depend on a layer that few people maintain and that untrusted input can reach.** Funding does not

buy control. It buys a voice, earlier warning and a healthier project. The xz utils backdoor, reported on 29 March 2024 and told in full in Chapter 8, showed what a thinly maintained compression library can carry into a whole distribution. Governments now fund this work too: Germany's Sovereign Tech Agency runs the Sovereign Tech Fund, which invests in the open software components that, in its words, underpin Germany's and Europe's competitiveness.

Fork means you take a copy of an open layer and carry your own changes on it. The rule: **fork only when you need changes the upstream will not take, and you can afford to keep importing upstream fixes for as long as you ship.** A fork costs little in its first year and more every year after, because upstream keeps moving. Google's BoringSSL, announced in June 2014, is the instructive case: after years of carrying patches on top of OpenSSL across several code bases, Google switched to importing changes from OpenSSL instead of rebasing on top of them, and said plainly it was not aiming to replace OpenSSL as an open source project.

Build means you write and own the layer. The rule: **build only when you have a reason to own the layer (differentiation, or continuity, sovereignty or cost) or no rentable option, when its residual security risk is explicitly assessed and acceptable (lower than renting, if security is the reason), and when you have a tested recovery time.** Cloudflare's Pingora proxy, told in Chapter 14, is a build that passed those gates. Few companies have that profile. Your product's own screens are the layer most companies should build; your release path and signing keys are the layers most should control (Chapter 14).

> **HOW IT GOT THIS WAY**
>
> The four choices are not new; the stakes are. BoringSSL in 2014, Edge in 2018 and Pingora in 2022 were three large companies giving three different answers. What has changed since is the clock: Chapter 11 shows exploit times falling from weeks to days, which gives the patch-time and recovery questions more weight than they had.

Layer	Windows	Linux	macOS
Kernel	Rent only: Microsoft updates it	Rent from a distribution, tune it, or fork it	Rent only: Apple updates it
Boot chain	Rent: PC maker firmware, Microsoft certificates	Rent shim and GRUB from a distribution; Microsoft signs shim	Rent only: Apple Boot ROM, LLB and iBoot
Compositor	Rent: DWM	Choose: Mutter, KWin or another; a fleet controls the choice	Rent: WindowServer
System web view	Rent WebView2, or bundle Chromium and control its patch path	Rent from the distribution, or bundle and control the patch path	On desktop macOS: rent WKWebView, or bundle and control the patch path
What you can own or control on all three	Own: your app and your approval screens. Control: your build and release path, your signing keys and your policy.		

These rules are for desktop macOS, not iOS. On iOS and iPadOS, apps use WebKit, except that as of November 2026 Apple allows other engines under strict entitlements for users in the EU (iOS 17.4 and later) and in Japan (iOS 26.2 and later).

The full Pixel Bill of Materials

Here are selected rows from Chapters 1 to 8, and the three AI rows from Chapter 13, in one place, as a starting inventory rather than a complete list: file systems, C runtimes, compilers, isolation runtimes and others belong on your copy too, with the final columns. The bill is split in two so it stays readable on a page. The first table answers **who**: who writes, pays for, updates and answers for each component. The second answers **how exposed**: the support clock, reachability, the patch time when a flaw is found first by someone else's AI, the tested recovery time, and the choice. The patch and recovery columns describe what typically sets the time; your own numbers belong in the blank copy at the end of this chapter.

Layer	Component	Who writes it	Who pays	Who can update it	Accountable owner
Silicon	CPU microcode	Intel, AMD; Apple for its chips	Built into the chip price	Chip maker, via PC maker or OS	Endpoint platform lead
Firmware	UEFI firmware	Firmware vendor plus PC maker	Built into the device price	PC maker only	Endpoint platform lead

Firmware	Secure Boot certificates	Microsoft; PC maker for the PK	Microsoft	Microsoft, via firmware and Windows updates	Endpoint platform lead
Kernel	NT, Linux or XNU kernel	Microsoft, upstream community, Apple	Vendor licenses; contributing companies	OS vendor or distribution; on Linux, also you	Infrastructure lead
Kernel	Endpoint security agent	Security vendor	You, by subscription	The vendor, often many times a day	Security operations lead
Drivers	GPU and network drivers	Hardware makers, Mesa, Apple	Hardware price	The maker or OS vendor	Endpoint platform lead
Security	Consent and elevation prompts	OS vendor or desktop project	OS vendor	OS vendor only	Security architect
Security	Your approval screens	You	You	You	Product owner of the flow
Runtimes	.NET, Node.js in Electron, Python	Vendor or open project	Vendor, sponsors, foundations	OS vendor, or every app that bundles a copy	Engineering lead per app
Graphics	System compositor	OS vendor or desktop project	OS vendor, distributions	OS or distribution vendor	Endpoint platform lead
Graphics	HarfBuzz, FreeType, image decoders	Small open teams	Mostly sponsors	Every product that ships a copy	Security architect
Browser	Chromium bundled in your app	Chromium project, Electron team	Google, sponsors, you	You, by shipping a release	Engineering lead per app
AI	Hosted model behind your agent or feature	The model provider	You, per use	The provider only	Product owner of the AI feature
AI	Agent permissions and tool list	Your team, on a rented framework	You	You	Security architect
AI	UI semantics agents read (labels, roles, accessibility tree)	Your app team, on the OS toolkit	You	You, inside the toolkit's limits	Product owner of the screens
Toolchain	Build pipeline and signing keys	Your team	You	You	Release engineering lead
Toolchain	Store or notarization path	Platform owner	Platform owner	Platform owner	Release engineering lead

Component	Support clock	Reachable from untrusted input?	Patch time when AI finds it first	Tested recovery time	Choice
CPU microcode	Chip maker's support list	Indirectly, as Spectre showed	Chip maker, then PC maker or OS, then you	Roll back through firmware or OS; drill it	Rent; track per model
UEFI firmware	PC maker's support for the model	Yes: LogoFAIL used a boot logo	Set by the PC maker, then your rollout	Re-flash per device; often a site visit	Rent; buy long support

Secure Boot certificates	2011 set expired June and October 2026	No, but they decide what boots	Microsoft and PC maker updates	Verify the 2023 certificates are enrolled (`db` and KEK) on every model, then test boot with media signed by the 2023 CA before rollout	Rent; keep the calendar
NT, Linux or XNU kernel	Release servicing dates	Yes	Vendor cycle; on Linux, can be yours	Reimage from a known-good image	Rent; Linux fleets may tune
Endpoint security agent	Your contract	Yes, by design	Vendor, with no change window	Boot to safe mode per device unless staged	Rent; demand staged rollout
GPU and network drivers	Maker's device support	Often	Maker, then OS vendor, then you	Pin and roll back the tested version	Rent; standardize models
Consent and elevation prompts	OS support life	Indirect: overlays and spoofed UI, not parsing	OS vendor only	Not yours; use a second channel	Rent
Your approval screens	Yours	Yes: overlays, capture, agents	Your next release	Your release rollback	Build; bind amount and payee
.NET, Node.js in Electron, Python	LTS windows of each project	Yes	One patch per bundled copy	Your app's rollback	Rent; track every copy
System compositor	OS release	Indirectly	OS vendor	Reimage	Rent
HarfBuzz, FreeType, image decoders	Project releases; your copy's age	Yes: fonts and images	Upstream, then every bundler separately	Each product's rollback	Rent and fund; map reachability
Chromium bundled in your app	Latest three Electron majors	Yes, if it loads outside content	Upstream, Electron, then your release	Your app's rollback	Rent; control the patch path
Hosted model behind your agent or feature	The provider's model retirement schedule	Yes: prompts, documents and pages it reads	The provider's clock; you can only limit what the model may do	Switch to a tested fallback model or turn the feature off; drill it	Rent; contract for notice and data terms
Agent permissions and tool list	Yours	Yes: every page and file the agent reads	Your next configuration change, often hours	Revoke the agent's identity and tools	Build the policy; rent the framework
UI semantics agents read	Your release cycle	Indirectly: agents act on what it says	Your release cycle; the OS vendor's for the toolkit	Your release rollback	Build; version it like an API
Build pipeline and signing keys	Yours	Yes, through dependencies	Yours	Rebuild and re-sign from clean sources; drill it	Control (build only the trust-critical pieces)
Store or notarization path	Platform rules	Indirectly	Platform review time	A fallback channel where allowed	Rent; keep a fallback

Two of the new columns deserve a word. **Patch time when AI finds it first** asks what happens when a flaw is found by an automated tool before the maintainer hears of it. That is no longer hypothetical: Chapter 13 tells how Google's Big Sleep agent found a SQLite flaw that attackers were preparing to use. The column asks how many hands a fix must pass through before it reaches your machines. **Tested recovery time** asks for a number from a drill. Owning source code does not mean you can rebuild, sign, deploy and roll back a working system after a supplier fails. Only a drill tells you.

Three companies, three bills

THE SOFTWARE MAKER

CHOICE	LAYERS	WHY
OWN	PRODUCT UI AND APPROVAL FLOWS	CUSTOMERS SEE THESE, AND ONLY THE COMPANY CAN FIX THEM
CONTROL (BUILD ONLY THE TRUST-CRITICAL PIECES)	BUILD PIPELINE ON RENTED TOOLS, SIGNING KEYS, UPDATE CHANNEL	EVERY FIX IT SHIPS PASSES THROUGH THEM
FUND	TEXT SHAPING AND IMAGE DECODING LIBRARIES IT SHIPS	THIN MAINTAINER TEAMS; REACHABLE FROM CUSTOMER FILES
FORK	NONE BY DEFAULT	NO UPSTREAM REFUSES CHANGES IT NEEDS
RENT	OS, KERNEL, COMPOSITOR, GPU DRIVERS, CHROMIUM THROUGH ELECTRON OR THE SYSTEM WEB VIEW	VENDORS PATCH THESE FASTER THAN A PRODUCT TEAM COULD

Its sharpest call is the browser engine on the desktop. If it bundles Chromium through Electron, it has rented the engine but controls the patch path: every Chromium fix reaches customers only when it ships a release. If it uses the system web view, it gives up control of the version and gains the vendor's patch speed. Either can be right. What is not right is choosing by habit and discovering the patch path during an incident.

THE REGULATED COMPANY

CHOICE	LAYERS	WHY
OWN	ACCESS POLICY, APPROVAL BINDING FOR PAYMENTS AND SIGNATURES, EVIDENCE TRAIL, INVENTORY	AUDITORS ASK FOR PROOF, AND ONLY IT CAN PRODUCE IT
CONTROL (BUILD ONLY THE TRUST-CRITICAL PIECES)	BUILD PIPELINE, SIGNING KEYS AND UPDATE SERVICE FOR ITS OWN APPLICATIONS	EVERY FIX TO ITS CLIENTS PASSES THROUGH THEM, AND AUDITORS ASK WHO CAN SIGN A RELEASE
FUND	OPEN COMPONENTS IT RELIES ON THROUGH ITS LINUX DISTRIBUTION	A VOICE AND EARLY WARNING ON ITS EXPOSURE
FORK	NONE	A FORK ADDS ITS OWN AUDIT SCOPE
RENT	OS, FIRMWARE, ENDPOINT SECURITY AGENT, CONSENT PROMPTS, SIGN-IN	RENT, WITH CONTRACT TERMS ON STAGED ROLLOUT AND SUPPORT DATES

It owns almost no code below its applications, yet it owns a great deal of responsibility. It controls its release path like the other two. The rest of its work is to rent well and prove it: staged rollout clauses for any vendor that writes to the kernel, a calendar of support dates, and approvals bound to signed data so that a dispute does not rest on a screenshot.

THE DEVICE FLEET OPERATOR

CHOICE	LAYERS	WHY
OWN	KIOSK SHELL AND PAYMENT SCREEN	CUSTOMERS TOUCH THESE AT EVERY SITE
CONTROL (BUILD ONLY THE TRUST-CRITICAL PIECES)	OS IMAGE (COMPOSED FROM VENDOR OR DISTRIBUTION PARTS AND PINNED), UPDATE AND ROLLBACK PATH	ONE BAD UPDATE REACHES EVERY SITE AT ONCE; IT MUST CONTROL THE ROLLOUT
FUND	THE DISTRIBUTION OR VENDOR SUPPORT FOR ITS LONG-LIFE IMAGE	ITS MACHINES OUTLIVE ORDINARY SUPPORT WINDOWS
FORK	NONE BY DEFAULT	COMPOSING AND PINNING A MINIMAL IMAGE IS NOT A FORK; FORK ONLY A COMPONENT WHOSE UPSTREAM WILL NOT TAKE A CHANGE IT NEEDS
RENT	FIRMWARE, KERNEL, GPU DRIVERS, COMPOSITOR, BROWSER ENGINE	PIN TESTED VERSIONS; DO NOT TAKE UPDATES UNTESTED

It is the one company of the three for which controlling more is often right, because its biggest cost is a site visit and its biggest risk is a fleet-wide failure. Composing and pinning the image, and controlling the rollback path, turns a fleet-wide outage into a rollback. It should still rent the kernel and the drivers, and pin them.

The Trusted Pixel: the sharpest call

Chapter 4 set out the Trusted Pixel threat models and a control for each. For the bill, the point is that the consent and payment pixel is a split decision. The OS vendor supplies and controls the system prompt, the secure desktop and the sign-in screen, and desktop systems offer third-party apps no general way to attest what the screen showed. You own the approval screen inside your product and what gets signed when the person presses the button. So **rent** the vendor's protected surfaces where they exist, **build** an approval flow that binds amount, payee and identity into the signed request, confirm actions that cannot be undone on a **second channel**, and decide on purpose whether agents acting for a user may approve at all.

> *You cannot own the glass. You can own what the signed request says the person approved.*

An owner for every seam

A **seam** is the place where one owner's layer hands off to another's. Incidents live there. In many organizations the pixel path is split across several teams: endpoint engineering owns the image and drivers, security owns the agent, application teams own the bundled engines, and a separate group owns the kiosk or the store listing. Each team can be competent and the seam can still belong to nobody.

The **accountable owner** column fixes that. It names one person, by role, for each row. That person does not do all the work. They answer three questions when the row fails: who do we call, what can they fix, and how long will it take. A row without an owner is a risk the company has accepted without deciding to.

Seam	Typically split between	Name one owner for
Firmware to OS	Hardware procurement, endpoint engineering	Support dates and certificate calendar
Kernel to security agent	Security operations, endpoint engineering	Staged rollout and recovery when an update fails
App to bundled engine	Each product team	Counting copies and shipping upstream fixes
App to consent screen	Product, security, legal	What is signed and what the person was shown
Image to field device	Field operations, endpoint engineering	Rollback without a site visit

SAMPLE CLAUSE: STAGED ROLLOUT (A TEMPLATE, NOT LEGAL ADVICE)

The seam between the kernel and a security agent failed at scale in July 2024 (Chapter 2). This is a plain-language starting point for counsel; the bracketed values are yours to set.

- **1. Stages.** Updates that run with kernel, firmware or administrator privilege, including content and configuration updates, ship in at least [three] stages: the supplier's own fleet, then no more than [1] percent of our devices, then the rest, at least [24] hours apart.
- **2. Our control.** We may delay or pause updates per device group. An emergency update needs prior notice to our named contact, with the reason.
- **3. Rollback.** The supplier halts a stage when [defined health signals] fail, and provides a remote rollback, tested [once a year].
- **4. Remedy.** A breach of clauses 1 to 3 gives us [service credits / a right to terminate].

The economics in plain words

Total cost of ownership (TCO) is everything a layer costs over its life, not just its price. For a rented layer it includes the license, the staff time to test and deploy updates, the incident work when it fails, and the memory and disk it takes on every machine. For an owned layer it adds the engineers, their security response, the build and signing infrastructure, and the cost of keeping a fork current. Compare the two over the life of the product, not over the first year, because forks and builds are cheapest at the start.

Switching cost is what it takes to move a layer to another owner: rewriting code, retraining people, retesting, and migrating data and devices. It is the price of a vendor's roadmap change. Microsoft paid it in 2018 when it moved Edge to Chromium. Your company pays it when a runtime, an engine or a device line is retired under you.

Exit options are the switching paths you prepare before you need them: a second supplier qualified, a contract clause on support dates and source escrow, a data export you have tested, a fallback distribution channel. An exit option is cheap to buy early and expensive to improvise. For every rented row with a high exit cost, write down the exit you would take, and test one part of it each year.

ILLUSTRATION: A 7-YEAR KIOSK FLEET, FIRMWARE CLOCK AGAINST STICKER PRICE

This is an illustration with stated assumptions, not data. A **device fleet operator** plans 2,000 check-in kiosks across 400 sites, to run for 7 years. Model A costs $900 and comes with 3 years of firmware updates. Model B costs $1,200 and comes with 7 years. Assume that if a firmware flaw needs a vendor fix after Model A's support ends, the operator replaces the fleet early, at $900 per kiosk plus one $400 site visit per site.

ITEM	MODEL A	MODEL B
PURCHASE, 2,000 KIOSKS	$1,800,000	$2,400,000
FIRMWARE UPDATES COVER	YEARS 1 TO 3	YEARS 1 TO 7
EARLY REFRESH IF A FIRMWARE FIX IS NEEDED IN YEARS 4 TO 7	$1,800,000 + 400 × $400 = $1,960,000	NOT NEEDED: THE VENDOR SHIPS THE FIX
BREAK-EVEN CHANCE OF THAT EVENT	$600,000 ÷ $1,960,000 = ABOUT 31 PERCENT	

If you judge the chance of at least one firmware flaw needing a vendor fix in years 4 to 7 to be above about 31 percent, the longer support clock is the cheaper choice. Firmware clocks do run out inside device lives: the 2011 Secure Boot certificates in the bill above expired in 2026. The model leaves out resale value, the benefit of newer hardware and the cost of running exposed while you decide, so put your own numbers in each cell. The method is the point: price the support clock, not just the sticker.

The same bill on a server-side stack

The method is not only for desktops and kiosks. A service that runs on Kubernetes in a public cloud has the same kinds of owners, with different names: the cloud's managed control plane plays the part of the OS vendor, the identity provider plays the part of the consent screen, and the key management service (KMS) holds keys the way your signing setup does. The "pixel" is the page or API response your customer receives. The rows below use the seven chapter columns; add the three extra columns of the full bill on your own copy.

Component	Who writes it	Who pays	Who can update it	Support clock	Reachable from untrusted input?	Your choice
Cloud control plane (managed Kubernetes API server)	The cloud provider, on upstream Kubernetes	You, per cluster hour	The provider; on Amazon EKS it upgrades the control plane itself when extended support ends	On EKS: 14 months of standard support per version, then 12 months of extended support at extra cost	Yes: the API faces your pipelines and people	Rent; upgrade on your schedule, before the forced one
Worker nodes and their OS image	A distribution or the provider	You, as compute	You, on EKS managed and self-managed node groups, which are not upgraded with the control plane; under EKS Auto Mode, AWS replaces nodes after a control-plane upgrade	The image's support dates	Yes, through the workloads they run	Rent the parts; compose and pin the image
Container base images and app dependencies	Open projects and vendors	Sponsors, vendors	You, by rebuilding and redeploying	Each project's releases	Yes: requests reach this code	Rent and fund; rebuild on every upstream fix
Identity provider (people and workloads)	The identity vendor	You, by subscription	The vendor	Your contract	Yes: sign-in faces the internet	Rent; own the policies and the break-glass accounts
Key management service	The cloud provider	You, per key and request	The provider for the service; you for key policy and rotation	The provider's service terms	Indirectly: through whoever can call it	Rent; own the key policy and drill a disabled-key day
Build, registry and deploy pipeline	Your team, on rented tools	You	You	Yours	Yes, through dependencies	Control (build only the trust-critical pieces; Chapter 14)

Two calendar facts show why the support clock column matters here too. The Kubernetes project supports each minor release for roughly fourteen months. Amazon EKS, as of November 2026, offers 14 months of standard support and 12 months of paid extended support per version, then upgrades the control plane on its own. AWS posts a notice of the end-of-support date at least 60 days ahead, but says it cannot give a time frame for the automatic upgrade and sends no notification before it. An in-place upgrade can be rolled

back within 7 days; an automatic one cannot. That is a rented layer whose owner will change it for you, on a date you can read today. Put it in the calendar, name an accountable owner, and measure how long a full cluster upgrade takes in a drill.

SECURITY LENS

Who can change it? Across the whole bill, most rows are changed by someone else: chip makers, PC makers, OS vendors, security vendors, open projects. You change your app, your approval flows, your policy, your build path and your keys. The bill makes that split visible, row by row.

How fast can it be fixed? As fast as the slowest hand in the chain. A fix in a shared decoder must pass from upstream to every product that bundles it; a firmware fix passes through the PC maker; your own code passes only through you. Owning a layer helps security where it removes hands from that chain, strips features or privilege you do not need, or keeps fixes coming after a supplier stops.

What does AI change here? It adds rows the old bill did not have: the hosted model, the agent's permissions and the UI semantics agents read (Chapter 13). And because a flaw may now be found first by someone else's tool, the patch-time and recovery columns are the ones to measure first.

BUSINESS IMPLICATIONS

Security. The **regulated company** gains most from the accountable owner column: an auditor's question about kernel code or payment approvals now has a named answer and a calendar. The **device fleet operator** gains most from tested recovery time, because its worst day is every site failing at once.

Cost. The **software maker** should compare TCO over the product's life before it bundles one more runtime or engine, since each copy brings its own patch path. Funding a few thin open libraries usually costs less than one incident in them.

Value. All three companies own the same short list, their product's screens and their approval binding, and control two more: their release path and their keys. That is where owning changes who can fix a defect, and how fast. Everything else is a rental to manage well.

The rule for the bill: no row without a named accountable owner and a support date.

Your blank Pixel Bill of Materials

Copy these two tables for your own product. Fill one row per component that sits between your code and your customer's screen. Start with the rows from this chapter, then add what is specific to you.

Layer	Component	Who writes it	Who pays	Who can update it	Accountable owner

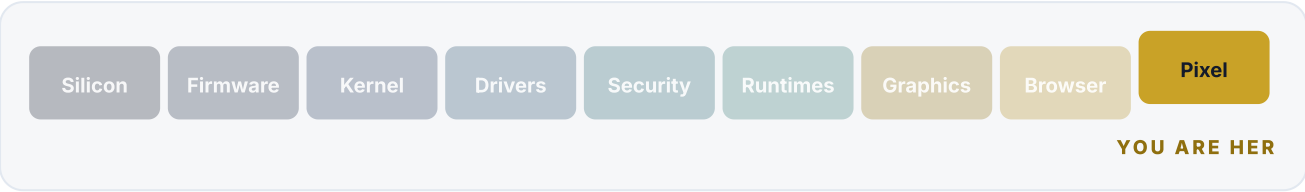
Component	Support clock	Reachable from untrusted input?	Patch time when AI finds it first	Tested recovery time	Choice

The Monday plan

1. **List the path and name its owners.** For your main product, write every component between your code and the customer's screen, on each system you ship to, and give each row one accountable owner by role.
2. **Put the dates in a calendar.** Support end dates, certificate expiry dates and contract renewals, each with a reminder a year ahead.
3. **Run one recovery drill.** Break the row whose failure would hurt most, in a test environment, and write the measured recovery time in the bill.
4. **Make one decision a month.** Run the nine questions on one row, choose rent, fund, fork or build, and take the decision and its reasons to your leadership team.

EPILOGUE

The Last Layer Is the Human



This book followed one key press from a switch in a keyboard to a lit dot. It stopped at the panel, because that is where the stack ends. The path does not end there. Every path from the bit to the pixel ends in a human eye, and the eye is a layer that no vendor ships, no update patches and no contract transfers.

The screen acts on a person

Light can harm, and the rules already say so. The Web Content Accessibility Guidelines, in success criterion 2.3.1, a Level A requirement, say content must not flash more than three times in any one second unless the flashes stay below set thresholds, because flashing can trigger seizures. Many displays dim by switching their light on and off very fast, pulse-width modulation, and TÜV Rheinland warns that such flicker can trigger headaches and migraines in sensitive people; it is chosen when someone buys the hardware, so it belongs on the procurement sheet. The medical alarm standard IEC 60601-1-8 sets how visual alarms are shown on monitors that run all night beside patients. And European law has treated the screen as a workplace since Council Directive 90/270/EEC of 29 May 1990, whose annex says that "the image on the screen should be stable, with no flickering or other forms of instability". For a **device fleet operator**, brightness at night, what an idle kiosk shows and whether anything flashes are design decisions with a person on the other side.

> **THE ONE ROW YOU CANNOT RENT**
>
> Every other row in the Pixel Bill of Materials can be rented, funded, forked or built. This one cannot. No bill of materials covers a retina. Duty of care is the one layer you cannot rent.

What "every pixel matters" means

The slogan on this series has meant three things in this book, and each one built on the last.

First, it meant **rendering control**: who draws your product. Between your code and the screen sit a runtime, a text engine, a browser engine, a compositor, a driver and a GPU, and each has an owner who can change what your customer sees.

Second, it meant **security**. Whoever can ship updates to those layers can change code on your customers' machines, and many of the layers can be reached by a file, a font, an image or a web page from outside. The pixel path is an attack path, and owning a layer changes who can fix it, and how fast.

Third, it meant **trust: what the person saw is what the system did**. A consent, a payment or a signature is only as good as the screen it was given on. The system can prove much about the transaction and little about the glass.

The epilogue adds the reason under all three. Every pixel matters because a person is looking at it.

> *Own what you can fix. Rent what you can trust. And never forget who is looking.*

APPENDICES

APPENDIX A

Glossary

These are the terms this book uses, in plain words. 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 C

Term	Meaning	Chapter
Access control list (ACL)	The list attached to each Windows object, such as a file or registry key, that says which users and groups may do what with it.	Ch 4
Access token	A Windows kernel object that says which user a process runs as, which groups it belongs to and which privileges it holds. The kernel checks it against the ACL.	Ch 4
Accessibility tree	The structured description of what is on screen (roles, labels, values) that a system exposes for screen readers. AI agents now read it too.	Ch 13
Accountable owner	The one person, named by role, who answers for a row of the Pixel Bill of Materials when it fails: whom to call, what they can fix and how long it takes.	Ch 15
APFS	Apple File System, the macOS default since High Sierra (2017). Built for flash storage, with encryption, snapshots and cheap file clones.	Ch 3
AppArmor	A Linux security module that confines programs by file path. Merged into the mainline kernel in Linux 2.6.36 (2010).	Ch 4
ArkWeb	The web engine of HarmonyOS, which its documentation says is developed on Chromium. Each OS release is tied to one Chromium milestone.	Ch 12
Blink	The rendering engine of the Chromium project, forked from WebKit by Google in April 2013.	Ch 7
Boot ROM	Boot code fixed in the chip when it is made. On Apple silicon it is the hardware root of trust, and it can never be updated.	Ch 1

Chromium	The open-source browser project, led by Google, whose Blink and V8 engines run inside Chrome, Edge, WebView2 and Electron.	Ch 7
CNA	CVE Numbering Authority: an organization allowed to assign CVE records. The Linux kernel project became one in February 2024.	Ch 11
Compositor	The system program that owns the whole screen and combines every window's buffer into one frame: DWM, Mutter or KWin, WindowServer.	Ch 6
Computer use	An AI model that reads screenshots, moves the cursor, clicks and types. Anthropic released it in public beta on 22 October 2024.	Ch 4, Ch 13
Container	A process with its own view of files, network and process IDs, built from kernel features. All containers on a host share one kernel.	Ch 8
Control	This book's word for deciding the composition, signing and rollout of parts you did not write, such as a composed and pinned OS image or your release pipeline on rented build services. You own a layer when you fork or build its code; you control it when you decide how it is assembled and shipped. A vendor controls what it supplies, such as the OS consent prompt. In the bills the choice reads "Control (build only the trust-critical pieces)".	Brief, Ch 11, Ch 14, Ch 15
CVE	Common Vulnerabilities and Exposures: a public record and identifier for one known security flaw, such as CVE-2023-4863.	Ch 6, Ch 11
Cyber Resilience Act	Regulation (EU) 2024/2847, in force since 10 December 2024. It requires a support period for products in scope of at least five years, unless expected use is shorter.	Ch 12

D to G

Term	Meaning	Chapter
db, dbx, KEK and PK	The Secure Boot key databases: trusted signers (db), revoked signers and images (dbx), the keys allowed to update them (KEK), and the PC maker's Platform Key (PK).	Ch 1
Decoder	Code that turns a file format, such as an image, a font or a video, into pixels or glyphs. It parses data from strangers, so attackers aim at it.	Ch 6
Decoder reachability	Whether untrusted bytes can reach a given decoder on a machine. Chapter 11 measures it with separate numbers: physical copies (C), installations (I), owners (O), reachable installations (R), the maximum lag in elapsed days (L) and copy-days (E).	Ch 6, Ch 11

Deemed export	Under US export rules (15 CFR 734.13), releasing controlled technology or source code to a foreign person inside the United States. Visual inspection, such as a screen share, can be a release (15 CFR 734.15(a)(1)).	Ch 4
Deepfake	Synthetic video or audio that imitates a real person. A January 2024 video call with deepfaked colleagues led to 15 transfers totalling HK$200 million.	Ch 4, Ch 13
DRM and KMS	The Linux kernel's Direct Rendering Manager, which manages GPU memory and jobs, and Kernel Mode Setting, which sets the display mode and flips finished frames.	Ch 6
Driver	The code that translates between the kernel and one piece of hardware. Traditionally it runs in the kernel, so one bad driver can stop the machine.	Ch 3
DriverKit	Apple's framework, introduced with macOS Catalina in 2019, for drivers that run in user space as driver extensions instead of in the kernel.	Ch 2, Ch 3
DWM	Desktop Window Manager, the Windows compositor. It has been always on since Windows 8.	Ch 6
Dynamic linking (PSD2)	The EU payment rule that the payer is shown the amount and payee, and that the authentication code is tied to them, so any change invalidates it.	Ch 4
eBPF	Small programs loaded into the Linux kernel at run time and checked by a verifier before they run, used for packet filtering and observability.	Ch 3
EEVDF	Earliest Eligible Virtual Deadline First, the Linux scheduler since Linux 6.6 (October 2023), which replaced the Completely Fair Scheduler.	Ch 2
eIDAS 2.0	Regulation (EU) 2024/1183, in force since 20 May 2024, which establishes the European Digital Identity Wallet.	Ch 4
Electron	A framework that bundles the Chromium engine and the Node.js runtime into each app's binary, so every Electron app carries its own engine.	Ch 5, Ch 7, Ch 11
Evergreen (WebView2)	The WebView2 mode in which all apps share one runtime that updates automatically, with the same updates as the Edge browser.	Ch 7
Exception levels	The Arm privilege levels: EL0 for applications, EL1 for the kernel, EL2 for a hypervisor and EL3 for the most trusted firmware.	Ch 1

Firmware	The program stored on a chip on the motherboard that wakes the hardware and hands control to an operating system loader.	Ch 1
Fixed Version (WebView2)	The WebView2 mode in which an app ships its own copy of the runtime, which is never updated unless the app ships a new one.	Ch 7
Fork upkeep	The recurring work of keeping a modified copy of an upstream project current: merging, rebuilding and retesting every upstream release, forever.	Ch 12, Ch 15
gVisor	A container sandbox from Google, written in Go, whose application kernel (the Sentry) handles a container's system calls itself.	Ch 8, Ch 14

H to M

Term	Meaning	Chapter
HarfBuzz	The open text shaping engine used by many browsers and Linux desktops.	Ch 6
Hybrid kernel	A kernel that borrows microkernel ideas but runs its services in kernel mode. Windows NT is usually called one; Apple calls XNU one.	Ch 2
Kernel	The core of the operating system, running with full hardware privilege. It runs processes, manages memory, schedules work and hosts drivers.	Ch 2
KEV	CISA's Known Exploited Vulnerabilities catalog: flaws the US cybersecurity agency knows are used in real attacks.	Ch 6, Ch 11
libwebp	The open library that decodes WebP images. Its flaw CVE-2023-4863 reached browsers, Electron apps and image libraries in September 2023.	Ch 6, Ch 7, Ch 11
LogoFAIL	Flaws, presented in December 2023, in the image parsers UEFI firmware uses to show a logo at boot. They run too early for Secure Boot to stop them.	Ch 1
Memory safety	A property of a language or program that prevents reads and writes outside the memory it should touch, such as past a buffer or after a free.	Ch 5
Microcode	Small internal programs that run many processor instructions. The chip maker can change it after sale, and it is loaded at every boot.	Ch 1

Microkernel	A kernel that keeps only a small core in kernel mode and runs other services as separate programs, so a failing service can be isolated, at the cost of more crossings between programs. Real safety and speed depend on the design.	Ch 2
MicroVM	A small virtual machine with its own kernel and only the devices it needs. AWS's Firecracker, announced in November 2018, is the main example.	Ch 8
MMU	Memory management unit: the hardware that translates each program's virtual addresses to real memory, using tables the kernel writes.	Ch 1
Monolithic kernel	A kernel that runs its core and its services, including file systems, networking and drivers, as one program in kernel mode. Linux is one, with loadable modules.	Ch 2

N to R

Term	Meaning	Chapter
OWASP Top 10 for LLM Applications	A ranked list of risks for applications built on language models. Prompt injection is first in both the 2025 and 2026 editions.	Ch 13
Patch lag	The days between an upstream fix and the moment the last device in your fleet runs it. Each owner in the chain adds to it.	Ch 11
Pixel Bill of Materials	This book's list of every component between your code and the customer's screen, with who writes it, who pays, who can update it, its support clock, its reachability and your choice.	Prologue, Ch 15
Pluton	A Microsoft security processor built into the CPU with AMD, Intel and Qualcomm, announced in November 2020. Its firmware is updated through Windows Update.	Ch 1
Prompt injection	Hiding instructions in content a model reads, such as a web page, an email or a document, so that the model follows them. OWASP says it is unclear whether any method prevents it fully.	Ch 13
Rasterization	Turning glyph outlines, or other shapes, into pixels at a given size.	Ch 6

Recall	A Windows feature on Copilot+ PCs, announced in May 2024, that saves regular snapshots of the screen so a user can search what they saw. It was redesigned to be opt-in and encrypted.	Ch 4, Ch 13
Release path	Your build pipeline, signing keys and update service: the route by which your code and every bundled fix reach customer machines. Its choice in every bill is Control: you run it on rented tools and build only the trust-critical pieces.	Prologue, Ch 11, Ch 14, Ch 15
Rent, fund, fork, build	The four choices for each layer: use it as shipped, pay its maintainers for a voice, carry your own copy, or write your own.	Prologue, Ch 15
Reproducible build	A build in which any party, given the same source, environment and instructions, can recreate bit-by-bit identical artifacts.	Ch 8, Ch 14
Ring 0 and ring 3	x86 privilege levels. The kernel runs in ring 0, the most privileged CPU mode, though hypervisors, firmware and hardware protections can still constrain it; applications run in ring 3.	Ch 1
Runtime	The code that must be on the machine for a program to run, though your team did not write it: a C library, a managed runtime such as the JVM or .NET, or a platform such as Electron.	Ch 5
Rust	A memory-safe systems language with no garbage collector and no virtual machine. It still has a small runtime of its own. Rust 1.0 was announced in May 2015.	Ch 2, Ch 5, Ch 14

S to T

Term	Meaning	Chapter
SBOM	Software bill of materials: a formal list of the components inside a piece of software. It says what is installed, not which code untrusted input can reach.	Ch 6, Ch 8
Seam	The place where one owner's layer hands work to another owner's layer. Incidents live there.	Ch 11, Ch 15
Seam tax	This book's name for the cost of a stack with many owners, counted as copies, update channels, patch lag and reachability.	Ch 11

Secure Boot	The UEFI feature that checks each boot loader's signature against the key databases before it runs. Required for Windows 8 certified PCs in 2012.	Ch 1
Secure desktop	A separate Windows desktop, used by default for UAC prompts, that only Windows processes can access. Other programs cannot draw on it or send it input.	Ch 4
Secure Enclave	A separate security processor on Apple chips, with its own boot ROM and operating system (sepOS), that holds keys and biometric data.	Ch 1, Ch 4
SELinux	A Linux security module that labels every process and file and checks each access against a central policy. Released by the NSA in December 2000.	Ch 4
Shaping	Turning characters into the right glyphs in the right order and position: joining Arabic letters, reordering Indic vowels, placing marks.	Ch 6
shim	The small first-stage Linux boot loader that Microsoft signs. It carries the distribution's key and checks GRUB, which checks the kernel.	Ch 1
SIP	System Integrity Protection: the macOS feature, introduced in 2015, that stops even the root user from changing protected system files.	Ch 4
Spectre and Meltdown	Processor flaws published on 3 January 2018 in speculative execution. Fixing them needed chip makers, kernels, compilers and browsers.	Ch 1
Support clock	The date when a component stops receiving updates, or a key or certificate expires.	Brief, Ch 1, Ch 15
TCC	Transparency, Consent and Control: the macOS layer that asks the user before an app may use the camera, microphone, protected folders or screen recording.	Ch 4
Tested recovery time	The measured time, from a drill someone actually ran, to rebuild, sign, deploy, roll back and restore a layer after a supplier fails.	Ch 8, Ch 14, Ch 15
Time-to-exploit	Mandiant's metric: "the average time taken to exploit a vulnerability before or after a patch is released", averaged over zero-days and n-days. Mandiant measured 5 days for 2023, excluding 15 outliers (47 with them).	Ch 11, Ch 13
Toolchain	Everything between a developer's idea and a running binary: compiler, editor and its assistants, build, packaging, signing, release and observation.	Ch 8, Ch 14

TPM	Trusted Platform Module: a security chip, or a protected function in the processor, that stores keys and records what booted. Windows 11 requires TPM 2.0.	Ch 1
Trusted Pixel	This book's name for applying the WYSIWYS principle to the whole pixel path: what the person saw must be what the system did. On desktop systems, third-party apps have no general way to attest what the screen showed; logs prove the transaction, not what the person saw.	Ch 4, Ch 15

U to Z

Term	Meaning	Chapter
UAC	User Account Control: since Windows Vista, it gives an administrator a filtered token for daily work and asks for approval before using the full one.	Ch 4
UEFI	Unified Extensible Firmware Interface: the firmware standard on PCs and servers today, maintained by the UEFI Forum since 2005.	Ch 1
UI semantics	The names, roles and order of controls on a screen. Because agents now depend on them, this book treats them as a versioned integration surface.	Ch 13
Update channel	Any path by which new code reaches a machine: the OS updater, a vendor service, an app store or an updater inside an app.	Ch 11
Update topology	The shape of the patch path: how many copies of a component sit on one machine, and who must act before each copy is fixed.	Ch 7
V8	Google's open-source JavaScript and WebAssembly engine, written in C++, used by Chromium, Node.js and Cloudflare's workerd.	Ch 7, Ch 14
Wayland	The Linux display protocol in which the compositor is the display server and apps hand it finished buffers. Version 1.0 was announced in October 2012.	Ch 6
WDDM	Windows Display Driver Model, introduced with Vista. It splits the GPU driver into user-mode and kernel-mode parts and can restart a hung driver.	Ch 3, Ch 6
WebGPU	A web interface that gives pages modern access to the GPU, on by default in Chrome 113. It is new attack surface that reaches the GPU driver.	Ch 7

WebKit	Apple's browser engine, which began as a fork of KDE's KHTML for Safari in 2003. On iOS outside the EU and Japan, every browser must use it.	Ch 7
Web view	A control that shows web content inside a native app window: WebView2 on Windows, WKWebView on Apple systems, WebKitGTK or Qt WebEngine on Linux.	Ch 7
Windows Resiliency Initiative	Microsoft's plan, announced in November 2024, that includes letting security products run outside kernel mode.	Ch 2
WOW64	The Windows layer that runs 32-bit applications on 64-bit Windows. It does not run 16-bit applications.	Ch 10
WYSIWYS	"What you see is what you sign": the established principle of electronic signatures, discussed by Landrock and Pedersen in 1998, that a signer must see exactly what is being signed. The Trusted Pixel applies it to approvals and payments; the book did not invent it.	Ch 4
XNU	The macOS kernel: a hybrid of the Mach kernel from Carnegie Mellon, components from FreeBSD and the IOKit driver API.	Ch 2
xz backdoor	A backdoor in the xz compression library's 5.6.0 and 5.6.1 release tarballs, reported on 29 March 2024 before it reached most stable distributions.	Ch 8
Zero-day	A flaw exploited before a fix exists. About 70 percent of the 138 flaws in Mandiant's 2023 analysis were zero-days.	Ch 11, Ch 13

APPENDIX B

Timeline Poster

Seventy years of the stack in two figures. Every date on this poster appears in a chapter of this book, where it is told in full and sourced. Only years with an event are drawn, so the vertical spacing is not to scale.

FIGURE B.1

1956 to 1998: one vendor, then a market of layers

	Operating systems	Browsers and runtimes	Security events	AI and security
1956	GM-NAA I/O on the IBM 704			No entries before 2017
1961	CTSS shown at MIT			
1964	IBM System/360			
1965	Multics papers			
1969	Unix begins at Bell Labs IBM unbundles software			
1972	IBM VM/370 announced			
1973	Unix rewritten in C			
1974	CP/M demonstrated			
1981	IBM PC with PC DOS			
1983		4.2BSD sockets		
1984			Trusting Trust lecture	
1985	Windows 1.0 Project Mach starts at CMU			
1986	Mach paper at USENIX			
1987	OS/2 announced, 1.0 ships X Window System Version 11	GCC 1.0		
1988		POSIX, IEEE 1003.1		
1990	Windows 3.0			
1991	Linux announced			
1992	Linux moves to the GPL Plan 9, first edition	UTF-8 designed for Plan 9		
1993	Windows NT 3.1	Mosaic released		
1995	Windows 95		PAM proposed, OSF RFC 86.0	
1996	Apple shelves Copland Apple to buy NeXT, not Be	Java 1.0		
1998	Intel Boot Initiative	Netscape opens its source		

From the first customer-written operating system to the PC stack and the free kernel. Details in Chapters 1 to 10 and 12.

FIGURE B.2

2000 to 2026: integration returns, and the clock speeds up

	Operating systems	Browsers and runtimes	Security events	AI and security
2000	IBM: $1 billion for Linux		NSA releases SELinux	
2001	Mac OS X 10.0 Be sells assets to Palm			
2002		.NET Framework		
2003	Xen paper at SOSP	Safari on WebKit		
2004		Firefox 1.0		
2005	Unified EFI Forum formed IBM withdraws OS/2			
2006	Amazon EC2 beta			
2007	iPhone; Android announced Vista: WDDM, UAC			
2008	App Store opens	Chrome beta		
2010	Windows Phone 7 unveiled systemd introduced		AppArmor merged	
2012	Wayland 1.0 Apple Maps apology		Secure Boot for Windows 8	
2013	Docker shown at PyCon	Blink forked from WebKit		
2014	Swift and Metal announced	BoringSSL fork of OpenSSL		
2015	Kubernetes 1.0 Windows Phone write-down Firefox OS phones stop	Rust 1.0	SIP in OS X El Capitan	
2016	Fedora 25: Wayland default Google reveals the TPU BlackBerry ends own phones			
2017	APFS default; AWS Nitro Linux runs all TOP500 Canonical ends Unity8			Rico screen dataset, UIST
2018	Firecracker; gVisor AWS Graviton	Edge moves to Chromium	Spectre and Meltdown	
2019	DriverKit; io_uring in 5.1 Huawei unveils HarmonyOS	WebAssembly a W3C standard	Huawei on the Entity List MSRC: 70% memory safety	
2020	Apple M1, unified memory Pluton announced	New Edge; WebView2 GA	BootHole in GRUB2	
2021		Edge Legacy support ends	EO 14028; SBOM elements Let's Encrypt root expires	UiPath lists (RPA)
2022	Rust support in Linux 6.1 BlackBerry 10 services end	Pingora described; workerd	NSA memory safety sheet	Copilot study: 55% faster
2023	Rust in win32k; EEVDF Azure Maia and Cobalt		libwebp CVE-2023-4863 LogoFAIL	
2024	Windows Resiliency plan	EU iOS: other engines Pingora open source	xz backdoor Kernel outage, 19 July Linux kernel becomes a CNA EU Cyber Resilience Act	Arup deepfake call DARPA TRACTOR Mandiant: 5-day exploit Computer use beta Big Sleep: SQLite flaw
2025	Rust experiment concluded GNOME 49: X11 session off Windows 10 Home, Pro end		FreeType CVE in KEV CISA, NSA: memory safety	Operator research preview AIxCC final at DEF CON GTG-1002 campaign report
2026			2011 boot certs expire	OWASP LLM Top 10 2026

"Kernel outage, 19 July" is the faulty security agent update told in Chapter 2. The Mandiant entry is its October 2024 analysis of 2023 data (Chapters 11 and 13). Red rings after 2010 are the owners that stopped, from Chapters 10 and 12. The 2011 Secure Boot certificates expired in June and October 2026 (Chapter 1).

KEY

How to read the poster

- Winner, platform or shipped system
- Failed system, or a step in its decline
- Research system, paper or study
- Security event, flaw or guidance

Shaded bands mark alternate decades. A teal dot with a red ring is research that lost its market.

Colors mark the kind of event; lanes mark the layer. Read across a row to see what happened in the same year on different layers.

The pattern in one paragraph

Read the operating system lane from top to bottom and the pendulum of Chapter 9 appears. One vendor owned the whole machine until 1969. From the 1970s to the 1990s the stack broke into layers, each one a market. From 2007, with the iPhone, and then with the cloud providers' own chips, integration came back. Read the two lanes on the right and the subject of Part III appears: security events and AI events crowd into the last decade, and they are why the time it takes your company to ship a fix now matters as much as who owns each layer.

APPENDIX C

Full Comparison Table

This appendix collects the three-system comparisons from every chapter into one reference, ordered from the silicon to the pixel and then up to the business layers. Each row says the same thing its chapter says; the chapter tells the story and gives the sources. Where a fact is time-sensitive, it is as of November 2026.

The key press, end to end (Prologue)

Step	Windows	Linux (Wayland desktop)	macOS
Key report	The keyboard sends a USB HID report that carries the modifier state and usage code 0x04, the "a and A" key position; the operating system applies the keyboard layout to decide the character		
Kernel input	kbdhid.sys, kbdclass.sys	usbhid, evdev	IOHIDFamily
Event to the app	win32k, WM_KEYDOWN and WM_CHAR	libinput, xkbcommon, Wayland key event	WindowServer, NSEvent
Text	DirectWrite	HarfBuzz shapes, FreeType draws	Core Text shapes and lays out; Core Graphics rasterizes
Compose the screen	Desktop Window Manager	Mutter or KWin	WindowServer, Core Animation
GPU and display	Direct3D, WDDM driver	Mesa, DRM and KMS	Metal
Light	The new frame is shown at the next display refresh		

Layer 1: Silicon and boot (Chapter 1)

Question	Windows PC	Linux PC	Mac with Apple silicon
Where the kernel runs	Ring 0 on x86; applications in ring 3	Ring 0 on x86; applications in ring 3	EL1 on Arm; applications in EL0
CPU and GPU memory	On a PC with a discrete graphics card, data is usually copied to its video memory; integrated GPUs share main memory	The same as Windows on the same hardware	Unified memory, one pool for CPU and GPU, since the M1 (2020)
First host firmware stage for OS boot	UEFI firmware from the PC maker, usually on an AMI, Insyde or Phoenix core	The same UEFI firmware	Boot ROM, fixed in the chip at manufacture
Who signs the next stage	Microsoft (Windows boot manager)	Microsoft signs `shim`; the distribution signs GRUB and the kernel	Apple: the Boot ROM checks the Low-Level Bootloader (LLB), which checks iBoot
Key and policy store	UEFI `db`, `dbx`, `KEK`, `PK`	UEFI keys, plus the Machine Owner Key list in `shim`	LocalPolicy, signed by the Secure Enclave
Hardware key holder	TPM 2.0 or Pluton	TPM 2.0	Secure Enclave, with its own boot ROM and sepOS
How microcode arrives	Mostly inside PC maker firmware updates; Microsoft has also published Intel microcode packages since March 2018	Loaded early by the kernel from the initial RAM disk, from distribution packages; late loading off by default since 5.19	Inside macOS updates
Who ships firmware fixes	The PC maker; some via Windows Update	The PC maker; some via the Linux Vendor Firmware Service	Apple, inside macOS updates

Layer 2: The kernel (Chapter 2)

Question	Windows (NT kernel)	Linux	macOS (XNU)
Design	Hybrid: kernel plus executive, drivers in kernel mode	Monolithic with loadable modules	Hybrid: Mach plus BSD plus IOKit
Who controls the source	Microsoft (closed)	Upstream maintainers, led by Linus Torvalds (GPLv2)	Apple (source published for parts of Darwin)
Scheduler model	Preemptive, 32 priority levels (0 to 31), round-robin within a level	EEVDF since Linux 6.6 (October 2023), replacing CFS	Mach threads, guided by app quality-of-service classes
Third-party code in the kernel	Common: drivers, security agents, anti-cheat	Common: modules from hardware and security vendors	Restricted: kernel extensions deprecated since 2019
Where outside code is being moved	User mode, under the Windows Resiliency Initiative	No single policy; upstream prefers drivers merged into the tree	System Extensions and DriverKit, in user space
Rust in the kernel	A Rust implementation of GDI regions (`win32kbase_rs.sys`) since Insider build 25905, July 2023	Supported since 6.1 (December 2022); the experiment declared concluded in December 2025	Not announced for the XNU kernel
Rust drivers in shipping products	Surface said in July 2025 it had shipped several Copilot+ PCs with drivers written in Rust	Linux 6.18 (November 2025) added the Rust Binder driver used by Android	Not announced
How drivers are built in Rust	`windows-drivers-rs`, an open-source Microsoft project	Rust bindings maintained inside the kernel tree	Not applicable
Who decides the pace	Microsoft	Kernel maintainers, subsystem by subsystem	Apple

Layer 3: Drivers, storage and network (Chapter 3)

Question	Windows	Linux	macOS
Driver interface	WDM, with KMDF and UMDF on top; stable across releases	No stable in-kernel interface; drivers belong in the kernel tree	DriverKit in user space; kernel extensions deprecated
Who writes most drivers	Hardware vendors	Vendors and kernel developers, upstream	Apple, plus vendors for add-on devices
Who lets a driver run	Microsoft signs kernel drivers (Windows 10, 1607 and later)	The distribution that builds the kernel, and the machine owner	Apple, through code signing and approval of system extensions
Fix path for a driver bug	Vendor ships it; Windows Update may carry it	Upstream patch, then each distribution's kernel	Apple's OS update, or the vendor's extension
GPU driver owner	The GPU maker, in two halves under WDDM	Kernel DRM driver plus Mesa in user space; NVIDIA's open modules ship outside the tree	Apple
Software fallback when the GPU fails	WARP	Mesa's llvmpipe	Documented only for the deprecated OpenGL framework, not for Metal

Feature	Windows	Linux	macOS
Default file system	NTFS	ext4 or XFS, by distribution; Btrfs on some	APFS
Checksums against silent corruption	ReFS (not bootable, as of November 2026)	Btrfs, for data and metadata	Not covered in this book
Snapshots	Volume Shadow Copy; ReFS file-level snapshots on Server 2022 and later	Btrfs, or the volume manager	APFS snapshots
Encryption at rest	BitLocker, below the file system	dm-crypt, below the file system	Built into APFS; FileVault ties the key to the user's password
Network interface for programs	Winsock, modelled on Berkeley sockets	BSD sockets; io_uring since Linux 5.1 (2019)	BSD sockets; Network.framework for app developers since 2018
Programs inside the kernel	eBPF for Windows, a work in progress as of November 2026	eBPF, checked by a verifier	Not covered in this book

Layer 4: Security, identity and the Trusted Pixel (Chapters 4 and 13)

Boundary	Windows	Linux	macOS
Identity of a process	Access token	User and group IDs, plus capabilities	User ID, plus code signature and entitlements
Who may touch an object	ACL on each object	Permission bits; SELinux or AppArmor policy	Permission bits; sandbox profile
Rules even admin cannot skip	Protected processes, driver signing	SELinux or AppArmor in enforcing mode	System Integrity Protection
Asking the person	UAC on the secure desktop	Desktop-specific prompts (polkit, portals)	TCC prompts
Service manager	Service Control Manager	systemd on most distributions	launchd
Sign-in framework	Windows Hello and credential providers	PAM	Touch ID matched in the Secure Enclave, through Authorization Services and LocalAuthentication; PAM for `sudo` and some services
Accessibility interface an agent can read	UI Automation	ATK and AT-SPI	AX API (NSAccessibility)
Who defines the control semantics	Microsoft's UI frameworks, then each app team	Toolkit projects such as GTK and Qt, then each app team	Apple's AppKit and SwiftUI, then each app team

How an app asks to capture the screen	Windows.Graphics.Capture: the user picks in system UI, and a yellow border marks what is captured	On Wayland desktops, the ScreenCast portal of xdg-desktop-portal	The Screen Recording permission under TCC

Layer 5: Runtimes and languages (Chapter 5)

Question	Windows	Linux	macOS
Base C runtime	Universal CRT, part of Windows 10 and later	glibc (musl on some small distributions)	libSystem
Native app framework	Win32, with .NET on top	None from the kernel; GTK or Qt chosen per app	Cocoa, with Swift and Objective-C
Managed runtime of the vendor	.NET CLR	None by default	None with a garbage collector; Swift uses reference counting
Who patches the base runtime	Microsoft, through Windows Update	The distribution vendor	Apple, through OS updates
Can an app override it?	Not the UCRT on Windows 10 and 11	Yes, with bundles or containers	Rarely, and only for older Swift targets

Layer 6: Graphics, text and the pixel (Chapter 6)

Part	Windows	Linux	macOS
Graphics interface	Direct3D 11 and 12	OpenGL, Vulkan (Mesa)	Metal (OpenGL deprecated since Mojave, 2018)
Compositor	DWM, always on since Windows 8	Mutter, KWin and others, under Wayland	WindowServer with Core Animation
Driver model	WDDM, since Vista	DRM and KMS in the kernel	Apple's own drivers
Text engine	DirectWrite	HarfBuzz shapes, FreeType rasterizes	Core Text shapes and lays out; Core Graphics (Quartz) rasterizes
Who ships fixes	Microsoft and the GPU maker	Kernel, Mesa, desktop project, distribution	Apple

Layer 7: The browser engine (Chapters 7 and 11)

Question	Windows	Linux	macOS
Vendor browser and engine	Edge: Chromium (Blink, V8)	No single vendor browser; the distribution chooses	Safari: WebKit
Web view an app can borrow	WebView2 (Chromium)	WebKitGTK (WebKit) or Qt WebEngine (Chromium)	WKWebView (system WebKit)
Who patches the borrowed engine	Microsoft, through the Evergreen runtime; or the app, with Fixed Version	The distribution's package maintainers	Apple, through system and Safari updates
Engine bundled in the app instead	Electron and similar frameworks: one Chromium copy per app, patched by each app maker		
Where extra engine copies come from	Each Electron app, and each Fixed Version WebView2 app	Each Electron app, and browsers that bundle their own libraries	Each Electron app, and each third-party browser
Who fixes a flaw in a shared library	Microsoft, for system libraries and the Evergreen WebView2 runtime; each product, for a Fixed Version runtime or another private copy	The distribution, once per source package that carries a copy	Apple for system copies; each vendor for its own
Who sets the engine roadmap	Google, with Microsoft as a large contributor	The upstream project of whichever engine	Apple

Layer 8: Toolchain and distribution (Chapters 8 and 11)

Question	Windows	Linux	macOS
Main native compiler	MSVC (Clang available)	GCC and LLVM/Clang	Apple Clang in Xcode
App package formats	MSIX, signed; older MSI and EXE installers	deb, rpm; Flatpak and Snap across distributions	App bundles and installer packages
Update channels the platform offers apps	Microsoft Store, WinGet, or the app's own updater	The distribution's repositories, Flatpak, Snap	Mac App Store, or the app's own updater
Who sits in the release path	The Store, if used; signing for MSIX	The distribution's maintainers, or the Flatpak or Snap store	Apple: the App Store, or notarization outside it
Who builds the OS from parts	Microsoft	The distribution, from the kernel community's kernel and thousands of projects	Apple

History and ecosystem (Chapters 9 and 10)

Question	Windows	Linux	macOS
Born in which swing	Horizontal: an OS licensed to many PC makers	Horizontal: a kernel owned by no single firm	Integrated: one firm, Mach and BSD under its own design
Who makes the hardware	Many PC makers	Anyone; the cloud providers on their own chips	Apple only
Who decides the next layer down	Intel, AMD, Qualcomm and the PC maker	The chip vendor and the distribution	Apple, since its own silicon
Direction today	Pulled toward integration: the Pluton security processor sits inside the CPU and is updated by Windows Update	Both: open upstream, integrated inside each cloud	Fully integrated, from chip to App Store
Who ships it on the machine	PC makers, at the factory	Some PC makers; mostly installed by IT, and the default on cloud servers	Apple, on every Mac
Where the apps come from	Direct downloads, installers and the Microsoft Store	Distribution repositories, Flatpak, Snap and containers	The Mac App Store and direct downloads
Compatibility promise	Strong: decades of Win32 apps, 32-bit through WOW64	Strong for the kernel's user interface; weaker for desktop libraries	Apple moves on when it changes chips or APIs
What it rents to others	Hardware and most apps	Almost everything: anyone can ship it	The app layer only

What a company can own (Chapter 15)

Layer	Windows	Linux	macOS
Kernel	Rent only: Microsoft updates it	Rent from a distribution, tune it, or fork it	Rent only: Apple updates it
Boot chain	Rent: PC maker firmware, Microsoft certificates	Rent shim and GRUB from a distribution; Microsoft signs shim	Rent only: Apple Boot ROM, LLB and iBoot
Compositor	Rent: DWM	Choose: Mutter, KWin or another; a fleet controls the choice	Rent: WindowServer
System web view	Rent WebView2, or bundle Chromium and control its patch path	Rent from the distribution, or bundle and control the patch path	On desktop macOS: rent WKWebView, or bundle and control the patch path
What you can own or control on all three	Own: your app and your approval screens. Control: your build and release path, your signing keys and your policy.		

HOW TO USE THIS TABLE

Read across a row to see who owns the same job on each system. Read down a column to see how many owners one system puts between your code and the screen. On macOS most rows end in Apple. On Windows most end in Microsoft or a hardware maker. On Linux most end in a distribution or an upstream project, which is why "which Linux?" is always the first question. Every row that ends in someone else is a row for your Pixel Bill of Materials.

APPENDIX D

Reading List and Sources

Part 1 is a short shelf of the papers and books that shaped the systems in this book. Part 2 lists the main primary sources behind each chapter, so a reader can check any claim at its origin. Every source in Part 2 was checked on 17 November 2026; web pages change, so check the date on anything you rely on.

Part 1: Foundational reading

Work	Year	Why read it
F. J. Corbató and V. A. Vyssotsky, "Introduction and Overview of the Multics System", Fall Joint Computer Conference	1965	The design goals of the "computer utility", many of which every modern OS still chases: protection, sharing and continuous operation.
E. W. Dijkstra, "The Structure of the 'THE'-Multiprogramming System", Communications of the ACM 11(5)	1968	The classic case for building an OS in layers, each relying only on the ones below. Short, and the root of the layer-by-layer view in this book.
D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", Communications of the ACM 17(7)	1974	The paper that made the case for a small, portable OS in numbers. Its process and file model still underlies Linux and macOS (Chapters 2 and 9).
K. Thompson, "Reflections on Trusting Trust", Communications of the ACM 27(8)	1984	Three pages on why reading source code does not prove what a binary does. The starting point for Chapters 8 and 14.
M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development", Summer USENIX Conference	1986	The microkernel that became part of XNU, the macOS kernel. Explains the trade-offs behind the word "hybrid" in Chapter 2.
F. P. Brooks Jr., The Mythical Man-Month: Essays on Software Engineering (Addison-Wesley)	1975	Lessons from leading OS/360. Still the best explanation of why owning a large layer costs more people and more time than planned (Chapters 9 and 12).
R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, free online at `pages.cs.wisc.edu/~remzi/OSTEP/`	Online, updated	The clearest modern textbook on processes, memory, concurrency and storage. Read it when a chapter of this book leaves you wanting the mechanism.
A. Agache et al., "Firecracker: Lightweight Virtualization for Serverless Applications", USENIX NSDI	2020	A recent, readable account of a company owning one narrow layer, a microVM monitor, and renting the kernel beneath it (Chapters 8 and 14).

Part 2: Main sources by chapter

Each entry gives the source, what it supports, and its address. The full list of claims and sources for each chapter is kept with the manuscript; these are the most important primary sources.

Reader's Brief

- **Microsoft, "Helping our customers through the CrowdStrike outage"** (20 July 2024): 8.5 million devices affected. `blogs.microsoft.com/blog/2024/07/20/helping-our-customers-through-the-crowdstrike-outage/`

Chapter 1, Silicon and Boot

- **Microsoft Learn, "User mode and kernel mode"**: one address space in kernel mode. `learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode`
- **Apple Newsroom, M1 announcement** (10 November 2020): unified memory. `apple.com/newsroom/2020/11/apple-unleashes-m1/`
- **Microsoft Learn, Secure Boot for OEMs**: db, dbx, KEK and PK. `learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot`
- **Microsoft Security blog, Pluton** (17 November 2020). `microsoft.com/en-us/security/blog/2020/11/17/meet-the-microsoft-pluton-processor-the-security-chip-designed-for-the-future-of-windows-pcs/`
- **Apple Platform Security, boot process for a Mac with Apple silicon**: Boot ROM, LLB, iBoot. `support.apple.com/guide/security/boot-process-for-a-mac-with-apple-silicon-secac71d5623/web`
- **Linux kernel documentation, microcode loading**. `kernel.org/doc/html/latest/arch/x86/microcode.html`
- **Google Project Zero, Meltdown and Spectre** (3 January 2018). `projectzero.google/2018/01/reading-privileged-memory-with-side.html`
- **CERT/CC note on LogoFAIL** (December 2023). `kb.cert.org/vuls/id/811862`
- **Microsoft Support KB5062710**: Secure Boot certificate expiration and CA updates. `support.microsoft.com/kb/5062710`

Chapter 2, The Kernel

- **CrowdStrike, technical details of the Falcon update**: Channel File 291, 04:09 to 05:27 UTC. `crowdstrike.com/en-us/blog/falcon-update-for-windows-hosts-technical-details/`

- **Apple XNU source repository**: hybrid kernel, Mach, FreeBSD and IOKit. `github.com/apple-oss-distributions/xnu`
- **Microsoft Learn, scheduling priorities**. `learn.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities`
- **Kernel Newbies, Linux 6.6**: EEVDF. `kernelnewbies.org/Linux_6.6`
- **Windows Experience blog, Windows Resiliency Initiative** (19 November 2024). `blogs.windows.com/windowsexperience/2024/11/19/windows-security-and-resiliency-protecting-your-business/`
- **Windows Experience blog, Windows security at Ignite 2026** (17 November 2026): the Windows endpoint security platform reaches general availability on Windows 11. `blogs.windows.com/windowsexperience/2026/11/17/windows-security-at-ignite-2026-start-secure-protect-what-matters-prepare-for-ai/`
- **Apple Developer, kernel extensions**: deprecation and replacements. `developer.apple.com/support/kernel-extensions/`
- **Windows Insider build 25905** (12 July 2023): Rust in win32k. `blogs.windows.com/windows-insider/2023/07/12/announcing-windows-11-insider-preview-build-25905/`
- **Check Point Research, Rust in the Windows kernel** (2025). `research.checkpoint.com/2025/denial-of-fuzzing-rust-in-the-windows-kernel/`
- **LWN, the Rust experiment concluded** (December 2025). `lwn.net/Articles/1050174/`

Chapter 3, Drivers, Storage and Network

- **Linux kernel documentation, "stable-api-nonsense"**. `kernel.org/doc/html/latest/process/stable-api-nonsense.html`
- **Microsoft, driver signing changes in Windows 10, version 1607**. `techcommunity.microsoft.com/blog/windowshardwarecertification/driver-signing-changes-in-windows-10-version-1607/364894`
- **Microsoft Learn, WDDM design guide**. `learn.microsoft.com/en-us/windows-hardware/drivers/display/windows-vista-display-driver-model-design-guide`
- **NVIDIA, open-source GPU kernel modules** (May 2022). `forums.developer.nvidia.com/t/nvidia-releases-open-source-gpu-kernel-modules/214128`
- **Microsoft Learn, ReFS overview**. `learn.microsoft.com/en-us/windows-server/storage/refs/refs-overview`
- **Apple Platform Security, APFS encryption**. `support.apple.com/guide/security/seca6147599e/web`

- **Apple WWDC 2018 session 715, Network.framework**. developer.apple.com/videos/play/wwdc2018/715/
- **eBPF for Windows repository**. github.com/microsoft/ebpf-for-windows
- **Cloudflare, "How we built Pingora"** (14 September 2022). blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/

Chapter 4, Security, Identity and the Trusted Pixel

- **Microsoft Learn, how User Account Control works**: the secure desktop. learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/how-it-works
- **Linux man page, capabilities(7)**. man7.org/linux/man-pages/man7/capabilities.7.html
- **Lennart Poettering, "Rethinking PID 1"** (April 2010). 0pointer.net/blog/projects/systemd.html
- **Commission Delegated Regulation (EU) 2018/389, Article 5**: dynamic linking. legislation.gov.uk/eur/2018/389/article/5
- **European Commission, eIDAS regulation**: digital identity wallets. digital-strategy.ec.europa.eu/en/policies/eudi-regulation
- **Android Developers, tapjacking**. developer.android.com/privacy-and-security/risks/tapjacking
- **Microsoft Learn, SetWindowDisplayAffinity**. learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowdisplayaffinity
- **Windows Experience blog, Recall security and privacy architecture** (27 September 2024). blogs.windows.com/windowsexperience/2024/09/27/update-on-recall-security-and-privacy-architecture/
- **15 CFR 734.13 and 734.15**: deemed export and release. law.cornell.edu/cfr/text/15/734.15

Chapter 5, Runtimes and Languages

- **Microsoft Learn, Universal CRT deployment**. learn.microsoft.com/en-us/cpp/windows/universal-crt-deployment
- **Swift.org, ABI stability and Apple platforms**. swift.org/blog/abi-stability-and-apple/
- **Node.js, previous releases and LTS policy**. nodejs.org/en/about/previous-releases

- **.NET support policy**. dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core
- **MSRC, "A proactive approach to more secure code"** (July 2019). microsoft.com/en-us/msrc/blog/2019/07/a-proactive-approach-to-more-secure-code
- **Google Security blog, memory safety in Android** (September 2024). security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html
- **CISA and partners, "The Case for Memory Safe Roadmaps"** (6 December 2023). cisa.gov/resources-tools/resources/case-memory-safe-roadmaps
- **CISA and NSA, memory safe languages guidance** (24 June 2025). cisa.gov/news-events/alerts/2025/06/24/new-guidance-released-reducing-memory-related-vulnerabilities
- **DARPA, TRACTOR announcement** (31 July 2024). darpa.mil/news/2024/memory-safety-vulnerabilities

Chapter 6, Graphics, Text and the Pixel

- **Microsoft Learn, Desktop Window Manager overview**. learn.microsoft.com/en-us/windows/win32/dwm/dwm-overview
- **Wayland 1.0 announcement** (22 October 2012). lists.freedesktop.org/archives/wayland-devel/2012-October/005967.html
- **GNOME 49 release notes for developers**. release.gnome.org/49/developers/
- **NVD record for CVE-2025-27363** (FreeType). nvd.nist.gov/vuln/detail/CVE-2025-27363
- **Chrome Releases, stable channel update of 11 September 2023**. chromereleases.googleblog.com/2023/09/stable-channel-update-for-desktop_11.html
- **Apple, macOS Mojave 10.14 release notes**: OpenGL deprecated. developer.apple.com/documentation/macos-release-notes/macos-mojave-10_14-release-notes
- **ITU-R Recommendation BT.2100-0** (July 2016). itu.int/rec/R-REC-BT.2100
- **Google Workspace Updates, canvas-based rendering in Docs** (May 2021). workspaceupdates.googleblog.com/2021/05/Google-Docs-Canvas-Based-Rendering-Update.html

Chapter 7, The Browser, the Second OS

- **Chromium blog, Blink announcement** (3 April 2013). blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html
- **Apple Newsroom, Safari unveiled** (7 January 2003). apple.com/newsroom/2003/01/07Apple-Unveils-Safari/

- **Windows Experience blog, Edge adopts Chromium** (6 December 2018). blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/
- **Microsoft Learn, WebView2 distribution**: Evergreen and Fixed Version. learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/distribution
- **Electron documentation and release timelines**. electronjs.org/docs/latest/tutorial/electron-timelines
- **GitHub advisory GHSA-j7hp-h8jx-5ppr** (libwebp). github.com/advisories/GHSA-j7hp-h8jx-5ppr
- **Chromium, site isolation design**. chromium.org/developers/design-documents/site-isolation/
- **Apple App Review Guidelines**, guideline 2.5.6. developer.apple.com/app-store/review/guidelines/
- **Apple Developer, alternative browser engines**. developer.apple.com/support/alternative-browser-engines/

Chapter 8, Toolchain, Distribution and the Edges of an OS

- **GCC releases**. gcc.gnu.org/releases.html
- **Microsoft Learn, MSIX overview**. learn.microsoft.com/en-us/windows/msix/overview
- **Apple Developer news, notarization for macOS Catalina** (3 September 2019). developer.apple.com/news/?id=09032019a
- **Andres Freund, xz backdoor report** (29 March 2024). openwall.com/lists/oss-security/2024/03/29/4
- **Tukaani project, xz backdoor notice**. tukaani.org/xz-backdoor/
- **AWS News blog, Firecracker** (26 November 2018). aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/
- **gVisor documentation**. gvisor.dev/docs/
- **Reproducible Builds, definition**. reproducible-builds.org/docs/definition/
- **NTIA, minimum elements for an SBOM** (12 July 2021). ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom

Chapter 9, Seventy Years of Operating Systems

- **Computer History Museum, GM-NAA I/O** (Robert L. Patrick). softwarepreservation.computerhistory.org/os/gm.html
- **Multicians, Multics history**. multicians.org/history.html

- **IBM, System/360 history**. ibm.com/history/system-360
- **Computer History Museum, IBM unbundling** (1969). computerhistory.org/revolution/mainframe-computers/7/172
- **Computer History Museum, fifty years of the PC operating system**. computerhistory.org/blog/fifty-years-of-the-personal-computer-operating-system/
- **AWS, the Nitro System journey**. docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/the-nitro-system-journey.html
- **Microsoft, Azure Maia and Cobalt** (15 November 2023). news.microsoft.com/source/features/ai/in-house-chips-silicon-to-service-to-meet-ai-demand/
- **Open Handset Alliance press release** (5 November 2007). openhandsetalliance.com/press_110507.html

Chapter 10, Why the Winners Won

- **Microsoft, history of Windows**. learn.microsoft.com/en-us/archive/blogs/uktechnet/history-of-windows
- **Plan 9 from Bell Labs, about**. 9p.io/plan9/about.html
- **Rob Pike, history of UTF-8**. cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt
- **Microsoft, restructuring of the phone hardware business** (8 July 2015). news.microsoft.com/2015/07/08/microsoft-announces-restructuring-of-phone-hardware-business/
- **Microsoft Learn, running 32-bit applications (WOW64)**. learn.microsoft.com/en-us/windows/win32/winprog64/running-32-bit-applications
- **Linux 0.12 release notes**: the move to the GPL. cdn.kernel.org/pub/linux/kernel/Historic/old-versions/RELNOTES-0.12
- **European Commission, Android decision** (18 July 2018). ec.europa.eu/commission/presscorner/detail/en/IP_18_4581
- **Apple Newsroom, iPhone SDK** (6 March 2008). apple.com/newsroom/2008/03/06Apple-Announces-iPhone-2-0-Software-Beta/

Chapter 11, The Seams, in Numbers

- **Electron release v26.2.1**: archive sizes and Chromium version. github.com/electron/electron/releases/tag/v26.2.1
- **libwebp fix commit and NEWS file**. github.com/webmproject/libwebp/blob/main/NEWS
- **Mozilla Foundation Security Advisory 2023-40**. mozilla.org/en-US/security/advisories/mfsa2023-40/

- **CISA Known Exploited Vulnerabilities catalog**. cisa.gov/known-exploited-vulnerabilities-catalog
- **Debian security tracker, CVE-2023-4863**. security-tracker.debian.org/tracker/CVE-2023-4863
- **CVE records for CVE-2023-4863 and CVE-2023-5129**. cveawg.mitre.org/api/cve/CVE-2023-4863
- **Mandiant, time-to-exploit trends 2023** (15 October 2024). cloud.google.com/blog/topics/threat-intelligence/time-to-exploit-trends-2023
- **Linux kernel documentation, CVEs**. docs.kernel.org/process/cve.html
- **Jerry Gamblin, 2024 and 2025 CVE data reviews** (independent counts). jerrygamblin.com/2026/01/01/2025-cve-data-review/

Chapter 12, The Price of Owning

- **Chromium, release cycle**. chromium.googlesource.com/chromium/src/+/main/docs/process/release_cycle.md
- **OpenHarmony documentation, web component overview** (ArkWeb). gitcode.com/openharmony/docs
- **BlackBerry, end of life for BlackBerry 10 services**. blackberry.com/us/en/support/devices/end-of-life
- **Mozilla, update on Firefox OS smartphones** (5 February 2016). blog.mozilla.org/press-uk/2016/02/05/update-on-firefox-os-smartphones-and-next-steps-for-connected-devices/
- **Canonical, "Growing Ubuntu for cloud and IoT"** (5 April 2017). canonical.com/blog/growing-ubuntu-for-cloud-and-iot-rather-than-phone-and-convergence
- **Apple, letter from Tim Cook on Maps** (28 September 2012). apple.com/letter-from-tim-cook-on-maps/
- **Federal Register, Huawei added to the Entity List** (21 May 2019). federalregister.gov/documents/2019/05/21/2019-10616/addition-of-entities-to-the-entity-list
- **Let's Encrypt, DST Root CA X3 expiration**. letsencrypt.org/docs/dst-root-ca-x3-expiration-september-2021/
- **Microsoft Learn, Windows 10 IoT Enterprise LTSC 2021 lifecycle**. learn.microsoft.com/en-us/lifecycle/products/windows-10-iot-enterprise-ltsc-2021
- **European Commission, Cyber Resilience Act**. digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act

Chapter 13, AI Moves the Seams

- **Anthropic, disrupting AI-orchestrated espionage** (13 November 2025). anthropic.com/news/disrupting-AI-espionage
- **OWASP, LLM01 prompt injection**. genai.owasp.org/llmrisk/llm01-prompt-injection/
- **OWASP, Top 10 for LLM Applications 2026**. genai.owasp.org/resource/owasp-genai-llm-top-10-2026/
- **Google Project Zero, "From Naptime to Big Sleep"** (1 November 2024). projectzero.google/2024/10/from-naptime-to-big-sleep.html
- **DARPA, AIxCC results** (August 2025). darpa.mil/news/2025/aixcc-results
- **Anthropic, computer use announcement** (22 October 2024). anthropic.com/news/3-5-models-and-computer-use
- **W3C, Core Accessibility API Mappings**. w3.org/TR/core-aam-1.2/
- **Microsoft Playwright MCP repository**. github.com/microsoft/playwright-mcp

Chapter 14, Controlling the Bit and the Toolchain

- **Microsoft Surface, "Safer drivers, stronger devices"** (July 2025). techcommunity.microsoft.com/blog/surfaceitpro/safer-drivers-stronger-devices/4431411
- **Kernel Newbies, Linux 6.18**: Rust Binder driver. kernelnewbies.org/Linux_6.18
- **Cloudflare, workerd open source** (27 September 2022). blog.cloudflare.com/workerd-open-source-workers-runtime/
- **Zed blog on GPUI** (7 March 2023): rendering "like a videogame". zed.dev/blog/videogame
- **Google Cloud, open-sourcing gVisor** (2 May 2018). cloud.google.com/blog/products/identity-security/open-sourcing-gvisor-a-sandboxed-container-runtime
- **Rust blog, Rust 1.0** (15 May 2015). blog.rust-lang.org/2015/05/15/Rust-1.0/
- **GitHub, research on Copilot productivity** (7 September 2022). github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/
- **METR, early-2025 study of experienced developers** (10 July 2025) and its February 2026 update. metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/
- **Daniel Stenberg, "Death by a thousand slops"** (14 July 2025). daniel.haxx.se/blog/2025/07/14/death-by-a-thousand-slops/

Chapter 15, What to Own, What to Rent

- **Sovereign Tech Agency**. sovereign.tech/
- **Adam Langley, "BoringSSL"** (20 June 2014). imperialviolet.org/2014/06/20/boringssl.html
- **Google, summer 2025 cybersecurity updates**: Big Sleep and CVE-2025-6965. blog.google/technology/safety-security/cybersecurity-updates-summer-2025/
- **Cloudflare, Pingora open source** (28 February 2024). blog.cloudflare.com/pingora-open-source/
- **CISA alert on xz utils** (29 March 2024). cisa.gov/news-events/alerts/2024/03/29/reported-supply-chain-compromise-affecting-xz-utils-data-compression-library-cve-2024-3094
- **Amazon EKS User Guide, updating the Kubernetes version of an EKS Auto Mode cluster**: Auto Mode replaces nodes after a control-plane upgrade. docs.aws.amazon.com/eks/latest/userguide/auto-upgrade.html

Epilogue, The Last Layer Is the Human

- **W3C, Understanding WCAG success criterion 2.3.1**. w3.org/WAI/WCAG22/Understanding/three-flashes-or-below-threshold.html
- **TÜV Rheinland, eye comfort certification brochure**. origin.tuv.com/content-media-files/master-content/services/products/1553-tuv-rheinland-eye-comfort-certification/tuv-rheinland-eye-comfort-certification_brochure_2023v2.pdf
- **EU-OSHA, Directive 90/270/EEC**: display screen equipment. osha.europa.eu/en/legislation/directives/5
- **Directive 90/270/EEC, Annex**. legislation.gov.uk/eudr/1990/270/annex
- **ASUS, flicker-free monitors**. asus.com/support/faq/1044285/

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 (this book)
Book 2	**From Code to Glass**: the rendering layer, from code to glass
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-bit-to-pixel.

Every Pixel Matters · Book 1 · From the bit to the pixel.