Made of Bugs

It's software. It's made of bugs.

Archives Subscribe Author

FEB 16, 2020

Computers can be understood

Introduction

This post attempts to describe a mindset I've come to realize I bring to essentially all of my work with software. I attempt to articulate this mindset, some of its implications and strengths, and some of the ways in which it's lead me astray.

Software can be understood

I approach software with a deep-seated belief that *computers and software systems can be understood.*

This belief is, for me, not some abstruse theoretical assertion, but a deeply felt belief that essentially any question I might care to ask (about computers) has a comprehensible answer which is accessible with determined exploration and learning.

In some ways, this belief feels radical today. Modern software and hardware systems contain almost unimaginable complexity amongst many distinct layers, each building atop each other. It is common — and substantially correct — to observe that no single human understands all of the layers in, say, a modern web application, starting from the transistors and silicon up through micro-architecture, the CPU instruction set, the OS kernel, the user libraries, the compilers, the web browser, and Javascript VM, the Javascript libraries, and the application code, not even to mention all the network services invoked in loading that code.

In the face of this complexity, it's easy to assume that there's just too much to learn, and to adopt the mental shorthand that the systems we work with are best treated as black boxes, not to be understood in any detail.

I argue against that approach. You will never understand every detail of the implementation of every level on that stack; but you *can* understand all of them to some level of abstraction, and any specific layer to essentially any depth necessary for any purpose.

Computers are not magic

At core, computers are built on a set of (mostly) deterministic foundations, which follow strict rules at each tick of the clock. We built layers upon layers of abstractions upon those foundations, each of which, as well, behaves in a (mostly) reproducible and deterministic way based on the abstractions at the previous level.

There is no magic. There is no layer beyond which we leave the realm of logic and executing instructions and encounter unknowable demons making arbitrary and capricious decisions. Most behaviors in one layer are comprehensible in terms of the concepts of the next layer, and all behaviors can be understood by digging down through enough layers.

Source, documentation, and reverse-engineering

In modern computer systems, a great many of the layers are open-source, and can be understood directly by reading the implementation. For a developer working on a Ruby on Rails app running against a MySQL database and deployed to a Linux server, every relevant piece of software involved is open-source and can be read if needed.

When systems aren't open-source, often there is deep and carefully-written documentation; in the above hypothetical system one of the first closed-source layers we will encounter is the hardware itself, likely a modern x86 processor. Intel makes available thousands of pages of documentation about their processors' interface that do an excellent job of exhaustively exploring virtually every detail of their hardware's behavior while executing code.

When source and documentation are unavailable or insufficient, systems can still be reverse-engineered via experiment and inspection by a sufficiently-determined engineer. Often, someone else has beat you to it and their results are available for you. Security engineers tend to be the most practiced at this particular skill; two of my favorite examples are Google Project Zero's reverse-engineering the Haswell microarchitecture's branch predictor, and the various efforts by security engineers to reverse-engineer and document macOS internals, such as this document on the WebKit heap in Safari.

Those two examples are massive efforts conducted by engineers at the top of their field with years of experience. I don't mean to imply that I could perform those reverse-engineering projects anywhere near as efficiently. For me, though, their existence proves (1) that this work *can* be done if I wanted to badly enough and (2) often I don't have to do it myself, if it has been done already and published.

Manifestations of this mindset

Understanding your dependencies

Perhaps the most direct manifestation of this mindset is that it leads me — and people who share something like this mindset — to spend time learning more about the systems they depend on, and how they work and how they are implemented. If I'm writing a non-trivial application against a library or framework, I almost always have a checkout of that library's source code on my laptop, and will regularly dig into it if I need to debug a strange behavior, or find that the documentation doesn't answer a question I have.

Debugging

Having this habit and knowing a lot about my dependencies has definitely been a superpower for debugging tricky bugs. If you work with any tool long enough, you will butt up against bugs in the tool which affect you, and it's valuable at a minimum to be able to accurately describe and diagnose them in terms of the tool's abstractions, in order to produce an actionable bug report or a minimal reproducer.

The trickiest bugs are often those that span multiple layers, or involve leaky abstraction boundaries between layers. These bugs are often impossible to understand at a single layer of the abstraction stack, and sometimes require the ability to view a behavior from multiple levels of abstractions at once to fully understand. These bugs practically require finding someone on your team who is comfortable moving between multiple layers of the stack, in order to track down; on the flip side, to the engineers with the habit of so moving around anyways, these bugs often represent an engaging challenge and form the basis of their favorite war stories.

One of my own favorite war stories of this genre was tracking down and identifying a single-bit memory flip on my desktop. This kind of problem can't even be fully understood without a good understanding of the interplay between userspace libraries, the kernel, the filesystem, and the hardware. My debugging war-story tumblr has a number of other great examples.

Documentation

Willingness and skill for reading source reduces your reliance on documentation. If the documentation is lacking in some way, you can always go to the source and look at the implementation for an authoritative answer.

The ability to answer questions this way is powerful to have on a team, since even the best documentation tends to have holes. It also has a downside, though; the engineers I've worked with (including myself) who are the most comfortable reading unfamiliar code bases are at risk of habitually undervaluing documentation (since they have gotten good at getting answers without it), and can be even worse at documenting their own system than the median engineer.

Security

Understanding security issues very often requires working at multiple levels of abstraction. An attacker attacking a system isn't bound by the documented or intended behavior of any layer; she cares about how the system *actually behaves in practice*, potentially including when one layer or input is "out-of-spec." The C specification says only that a buffer overflow is "undefined behavior"; understanding how to turn one into remote-code-execution, or reasoning about countermeasures like ASLR or DEP requires a deep understanding of the actual implementation of the abstract C specification by the compiler, libc, underlying hardware, and more.

I started my career substantially in security, while working at Ksplice, which sold zero-downtime software updates for the Linux kernel, primarily as a security feature. I learned a lot of my current skill and comfort with digging deeper into the stack and understanding all the layers of abstraction while spending time around a lot of security bugs there.

Performance

Understanding and reasoning about software performance also often involves understanding multiple layers of your stack. It's hard to write efficient Python code without some understanding of the CPython (or PyPy) implementation, and you can't write cache-efficient C code without some understanding of the generated code and the underlying hardware. Dig into a crowd of performance engineers, and you'll virtually always find a handful of engineers with the habit of always digging deeper to continually better-understand ever-more layers of abstraction.

Building mental models

A deeply related habit to trying to learn about the underlying layers of a software stack is the habit of trying to understand software by building detailed mental models of the underlying system. Instead of understanding systems (languages, libraries, APIs, etc) solely as collections of rules and behaviors and edge-cases, I try to build a smaller model of their core primitives, and the rules or principles that generate the larger behaviors of the system.

As a concrete example, I've written more `bash` shell scrips in my life than I should perhaps be proud of. At some point, instead of continually memorizing specific patterns and anti-patterns that happen to work or not work (when do you or don't need to quote something?), I stepped back to read the bash documentation in order to understand the various expansion phases that bash follows when processing a command line, and which ones are applied in which order in which context. This knowledge didn't eliminate the need to learn a ton of trivia to write shell scripts — arguably, it added more trivia — but having a framework to fit knowledge into both made it easier to retain that trivia, and increased its explanatory power in the face of novel problems or patterns of code.

I think there's a related belief here which ties into and is reinforced by the basic belief that computers are comprehensible: Computer systems tend to be, well, systematic, and have some core algebra or logic that is comprehensible, which is smaller than a complete list of possible behaviors, and which generates or at least organizes all of those behaviors. And, as an addendum, I tend to believe that work invested in learning these underlying systems will pay off in terms of understanding and working with the system.

Single-shot debugging

As a corollary of having good mental models of software systems, and of those systems being mostly deterministic, it becomes possible to make fairly detailed inferences about program state and behavior off of a a small number of observations about its behavior at points in time (perhaps a stack trace, or a log line, or a core dump). In the most extreme examples, a developer can sometimes root-cause a buggy behavior based on a single encounter with a bug. With a rich mental model of the system and the code at hand, you can perform backwards reasoning in the form of deductions like "Ah, if this field is set to NULL, someone must have set it … the only code that sets that field is {here}, {here}, and {here} … only the first and third could ever be called with a NULL argument …" and so on.

Even if you can't "one-shot" a bug, there's a general skill here of being able to formulate theories and hypotheses and refine your mental models based on observations of the system, which allow you to ask much more specific questions, which you can then test (in a debugger, with a print statement, by reading code, …), which then further refine your models. A rich mental model and the ability to play it forward and backwards in time is an incredible aid to debugging and to learning a system.

Single-shot debugging in kernel engineering

The more complex and nondeterministic a system is, the harder it gets to reliably predict behavior from a small number of observations, which I think in part explains the modern trend of "observability" in distributed systems — you need much more data to fully explain these systems' behaviors.

However, at the bottom of the stack, among systems engineers and especially kernel engineers, this skill set is widespread. I've seen threads on the LKML where a developer will post a single crash log with a stack trace and a register dump, and a handful of senior kernel lieutenants will collaboratively go on the hunt for the bug, making detailed inferences based on mapping between the register dump and the compiled code, and based on their understanding of all the places in the kernel that deal with the implicated data structures.

When I worked at Oracle (following the Ksplice acquisition), I talked with some Solaris kernel engineers there, and learned that they had taken this kind of approach even further. They apparently had an explicit goal of a 100% rate of root-causing kernel crashes based on a *single crash report*. In order to strive for this goal they had built a lot of elaborate crash-reporting and debugging technology — it wasn't just raw thinking hard about bugs — but at root I think this goal comes from the deep belief that their system, while complex, is understandable and mostly deterministic, and that they have the ability to reason about it. I found this story pretty inspiring.

I think in many ways, kernel development is the prime audience for this mindset. For one, it's not uncommon, especially a decade ago prior to our modern virtualization era, that your only ability to debug a kernel crash is by inspecting a crash dump or log trace — you may not have a debugger or even the ability to continue executing at all. And, for another, because the OS kernel is essentially the software closest to the hardware, the number of layers you have to understand to fully explain your code's behavior and interactions is comparatively small. You (mostly) only need to understand your C compiler / compiled code, and the hardware itself. Anyone developing in userspace (atop the kernel) has strictly more layers to work through.

Pitfalls of this mindset

I want to include a note here about the pitfalls of this mindset, and some places where I've observed it leading me astray. Overall, the belief in the fundamental comprehensibility of software, and the pursuit of detailed mental models has served me quite well, but I want to be clear that I don't think it's the only valid or useful approach, and that it comes with its own weaknesses.

The need to understand

A belief in the understandability of software systems can very easily become a **need** to understand the systems you work with. I have become very uncomfortable working in software systems where I don't have a good model of the underlying layers, and this discomfort can sometimes be harmful to accomplishing my goals.

I find it very hard to get started working with a complex system by just following a tutorial or two and performing small edits or local exploration out from that example. I am uncomfortable until I **understand** the roles and relationships of all the components I'm interacting with, at least at a high level.

Concretely, the other day I was attempting to stand up a single HTTP endpoint backed by Amazon Lambda (which I had never worked previously used). There are a million tutorials about this task, both from AWS and others. I'm quite confident that if I'd taken almost any of these and adopted it via trial-and-error I could have accomplished my task in something like 30 minutes. However, instead, I stubbornly insisted on starting from scratch and understanding each component I needed. Since performing any task on AWS involves stitching together approximately 15 different mind-numbingly-complex products, I soon ended up with 30 documentation tabs open, an endpoint that returned a server error no matter what I tried, and still no particularly better idea what the heck was going on. I eventually gave up and decided that the problem wasn't that important to solve anyways.

I do believe that if I had had more time and patience, I would eventually arrive at a fairly deep conceptual understanding of Lambda and the adjacent AWS products, and be much better equipped to debug my deployment or solve future problems. However, that wasn't my goal; I just wanted something that worked, within a time budget. And so, I instead just ended up without anything that worked, and without much of a better understanding of anything, either. The need to understand can be seductive and harmful when working atop a complex system, where your problem does not fundamentally require understanding the whole thing.

Do the easy thing first

I've lost track of the number of times that, faced with a bug in a dependency, I've spent days digging deeply into the dependency in order to identify and isolate the bug … only to realize that the bug was already fixed upstream, and we were pinned to an older version. Or when I've spent time trying to debug a crash in a binary that was built without debug symbols, by carefully poring through a coredump and x86 disassembly, only for a coworker to find a debug build, reproduce the issue there, and traipse through the green pastures of a working `gdb` session[1].

I have a particular set of software and systems skills, which happen to include binary reverse-engineering and rapidly coming up to speed on unfamiliar code bases. I have these skills in part because of my obsessive desire to understand the systems I work with. However, having these skills doesn't mean they're always the right skills to apply to a problem!

It's nearly always worth trying the easier approach first (upgrading a dependency, reaching for the debugger, a few passes of trial-and-error cargo-culting from working examples), and only reach for the big guns if those tools fail you.

Start with curiosity

I want to close with some thoughts about how to get started with this mindset, and how to begin acquiring concrete skill understanding unfamiliar computer systems. I've learned my toolkit over (at this point) about two decades of wrangling computer systems, and I worry about inadvertently presenting a story that computers can be understood, but only if you have my particular depth of experience doing so. While I can only confidently write from my own experiences, I do deeply believe the mindset discussed here has value no matter how experienced you are.

My advice for a practical upshot from this post would be: cultivate a deep sense of curiosity about the systems you work with. Ask questions about how they work, why they work that way, and how they were built. Ask yourself questions like "How would I have built this library?," identify the gaps where you don't know the answer, and add them to your mental backlog of topics to learn.

For an even more tactical takeaway, I might start with this: Read the source of your dependencies, if that's not already a habit you have. Do you write webapps on React? Try grabbing a checkout and reading through the source sometime. Are you working on a Django or Rails webapp? Check out the source of the framework in question. Even grab a copy of the Python or Ruby implementation, and take a look inside. Much of the standard library for those languages is written in the language itself, so you can even get started without learning much or any C. Your goal needn't be to understand all of it at once, or even ever — just to build your understanding, and your confidence that you can always understand more tomorrow.

Learning more about software systems is a compounding skill. The more systems you've seen, the more patterns you have available to match future systems against, and the more skills and tricks and techniques you develop to apply to future problems. Understanding immensely complex systems may seem out of grasp at first, but the more you try the easier it becomes.

One of my favorite examples of an engineer who publicly models this mindset is Julia Evans, who I was fortunate enough to work with at Stripe. She is incredibly curious about how computers work, and does an amazing job writing and talking, not only about what she learns, but how she learned it, and conveying a raw sense of curiosity and excitement and discovery. Some of my favorite examples:

- Her post about how she got into kernel development is a great concrete example of taking a scary area and finding a way in.
- Her talk on how to become a wizard mirrors many of the ideas in the post, and also comes with practical advice on how to implement them.
- Her post on asking great questions is an excellent resource for anyone working with others, or just who's curious about learning more in computing!

In closing

Computers are complex, but need not be mysteries. Any question we care to ask can, in principle and usually even in practice, be answered. Unknown systems can be approached with curiosity and determination, not fear.

These beliefs are a huge aspect of how I — and many other experienced engineers I know — approach software engineering. I hope this post serves as a useful articulation of this mindset. If it encourages you to reach out and try to learn or understand something new, drop me an email and let me know!

---

1. To give a specific example, I got this feedback from a number of people about my post about isolating an eventmachine memory leak a few years ago. They criticized me for jumping straight to spelunking through raw memory images, instead of trying a tool like Valgrind or tcmalloc's leak detector. In this case, I was successful in finding the bug and I'm not actually certain if those tools would have worked, but I think the point stands that my instincts are overly focused on trying the likely-harder approach first. ↩

Subscribe to my newsletter (blog updates and occasional other content):

email@domain.com Subscribe

Powered by Buttondown.

« Reflections on software performance Systems that defy detailed understanding »
