CHAPTER 15
Investigating Systems

By Pete Nuttall, Matt Linton, and David Seidman
with Vera Haas, Julie Saracino, and Amaya Booker

Most systems eventually fail. Your ability to investigate a complex system relies on a number of factors: in addition to having access to adequate logs and information sources for debugging, you need proper expertise. You also need to design your logging systems with protection and access control in mind. In this chapter, we walk through debugging techniques and provide some strategies for what to do when you're stuck. We then discuss the differences between debugging a system issue and investigating a security concern, and examine tradeoffs to take into account when deciding which logs to retain. Finally, we look at how to keep these valuable sources of information secure and reliable.

In an ideal world, we would all build perfect systems, and our users would have only the best of intentions. In reality, you'll encounter bugs and need to conduct security investigations. As you observe a system running in production over time, you'll identify areas for improvement and places where you can streamline and optimize processes. All of these tasks require debugging and investigation techniques, and appropriate system access.

However, granting even read-only debugging access creates a risk that this access may be abused. To address this risk, you need proper security mechanisms in place. You also need to strike a careful balance between the debugging needs of developers and operations staff, and the security requirements of storing and accessing sensitive data.

In this chapter, we use the term *debugger* to mean a human who is debugging software problems—not GDB (the GNU Debugger) or similar tools. Unless otherwise noted, we use the term "we" to refer to the authors of this chapter, not Google as a whole.

From Debugging to Investigation

> [T]he realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.
>
> —Maurice Wilkes, Memoirs of a Computer Pioneer (MIT Press, 1985)

Debugging has a bad reputation. Bugs surface at the worst of times. It can be hard to estimate when a bug will be fixed, or when a system will be "good enough" to let many people use it. For most people, it's more fun to write new code than to debug existing programs. Debugging can be perceived as unrewarding. However, it's necessary, and you may even find the practice enjoyable when viewed through the lens of learning new facts and tools. In our experience, debugging also makes us better programmers, and reminds us that sometimes we're not as smart as we think we are.

Example: Temporary Files

Consider the following outage, which we (the authors) debugged two years ago.[1] The investigation began when we received an alert that a Spanner database was running out of storage quota. We went through the process of debugging, asking ourselves the following questions:

1. What caused the database to run out of storage?

   Quick triage indicated the problem was caused by an accumulation of many small files being created in Google's massive distributed filesystem, Colossus, which was likely triggered by a change in user request traffic.

2. What was creating all the tiny files?

   We looked at service metrics, which showed the files resulted from the Spanner server running low on memory. According to normal behavior, recent writes (updates) were buffered in memory; as the server ran low on memory, it flushed the data to files on Colossus. Unfortunately, each server in the Spanner zone had only a small amount of memory to accommodate updates. As a result, rather

1 Although the outage occurred in a large distributed system, people who have maintained smaller and self-contained systems will see a lot of similarities—for example, in outages involving a single mail server whose hard drive has run out of space!

than flushing a manageable number of larger, compressed files,[2] each server flushed many tiny files to Colossus.

3. Where was the memory being used?

   Each server ran as a Borg task (in a container), which capped the memory available to it.[3] To determine where within the kernel memory was used, we directly issued the `slabtop` command on the production machine. We determine that the directory entry (dentry) cache was the largest user of memory.

4. Why was the dentry cache so big?

   We made an educated guess that the Spanner database server was creating and deleting vast numbers of temporary files—a few for each flush operation. Each flush operation increased the size of the dentry cache, making the problem worse.

5. How could we confirm our hypothesis?

   To test this theory, we created and ran a program on Borg to reproduce the bug by creating and deleting files in a loop. After a few million files, the dentry cache had used all the memory in its container, confirming the hypothesis.

6. Was this a kernel bug?

   We researched expected behavior of the Linux kernel, and determined that the kernel caches the nonexistence of files—some build systems need this feature to ensure acceptable performance. In normal operation, the kernel evicts entries from the dentry cache when the container is full. However, because the Spanner server repeatedly flushed updates, the container never became full enough to trigger evictions. We addressed this issue by designating that temporary files didn't need to be cached.

The debugging process described here illustrates many of the concepts we discuss in this chapter. However, the most important takeaway from this story is that *we debugged this issue*—and you can, too! Solving and fixing the problem didn't require any magic; it just required slow and structured investigation. To break down the characteristics of our investigation:

2 Spanner stores data as a Log-Structured Merge (LSM) tree. For details on this format, see Luo, Chen, and Michael J. Carey. 2018. "LSM-Based Storage Techniques: A Survey." arXiv preprint arXiv:1812.07527v3.

3 For more on Borg, see Verma, Abhishek et al. 2015. "Large-Scale Cluster Management at Google with Borg." *Proceedings of the 10th European Conference on Computer Systems*: 1–17. doi:10.1145/2741948.2741964.

- After the system showed signs of degradation, we debugged the problem using existing logs and monitoring infrastructure.
- We were able to debug the issue even though it occurred in kernel space and code that the debuggers had not seen before.
- We'd never noticed the issue before this outage, even though it had likely been present for several years.
- No part of the system was broken. All parts were working as intended.
- The developers of the Spanner server were surprised that temporary files could consume memory long after the files had been deleted.
- We were able to debug the kernel's memory usage by using tools provided by the kernel developers. Even though we'd never used these tools before, we were able to make progress relatively quickly because we were trained and well practiced in debugging techniques.
- We initially misdiagnosed the bug as a user error. We changed our minds only after examining our data.
- By developing a hypothesis and then creating a way to test our theory, we confirmed the root cause before we introduced changes to the system.

Debugging Techniques

This section shares some techniques for systematic debugging.[4] Debugging is a skill that you can learn and practice. Chapter 12 of the SRE book offers two requirements for successful debugging:

- Know how the system is supposed to work.
- Be systematic: collect data, hypothesize causes, and test theories.

The first of these requirements is trickier. Take the canonical example of a system built by a single developer who suddenly leaves the company, taking all knowledge of the system with them. The system may continue to work for months, but one day it mysteriously breaks and no one can fix it. Some of the advice that follows can help, but there's no real substitute for understanding the system ahead of time (see Chapter 6).

4 You may also be interested in the blog post "What Does Debugging a Program Look Like?" by Julia Evans.

Distinguish horses from zebras

When you hear hoofbeats, do you first think of horses, or zebras? Instructors sometime pose this question to medical students learning how to triage and diagnose diseases. It's a reminder that most ailments are common—most hoofbeats are caused by horses, not zebras. You can imagine why this is helpful advice for a medical student: they don't want to assume symptoms add up to a rare disease when, in fact, the condition is common and straightforward to remedy.

In contrast, given a large enough scale, experienced engineers will observe both common *and* rare events. People building computer systems can (and must) work to completely eliminate all problems. As a system grows in scale, and its operators eliminate common problems over time, rare problems appear more frequently. To quote Bryan Cantrill: "Over time, the horses are found; only the zebras are left."

Consider the very rare issue of memory corruption by bit flip. A modern error-correcting memory module has a less than 1% chance per year of encountering an uncorrectable bit flip that can crash a system.[5] An engineer debugging an unexpected crash probably won't think, "I bet this was caused by an extremely unlikely electrical malfunction in the memory chips!" However, at very large scale, these rarities become certainties. A hypothetical cloud service utilizing 25,000 machines might use memory across 400,000 RAM chips. Given the odds of a 0.1% yearly risk of uncorrectable errors *per chip,* the scale of the service could lead to 400 occurrences annually. People running the cloud service will likely observe a memory failure every day.

Debugging these kinds of rare events can be challenging, but it's achievable with the right kind of data. To provide one example, Google hardware engineers once noticed that certain RAM chips failed much more often than expected. Asset data allowed them to track the source of the failing DIMMs (memory modules), and they were able to trace the modules to a single provider. After extensive debugging and investigation, the engineers identified the root cause: an environmental failure in a clean room, in a single factory where the DIMMs were produced. This problem was a "zebra"—a rare bug visible only at scale.

As a service grows, today's strange outlier bug may become next year's routine bug. In the year 2000, memory hardware corruption was a surprise for Google. Today, such hardware failures are routine, and we plan for them with end-to-end integrity checking and other reliability measures.

5 Schroeder, Bianca, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. "DRAM Errors in the Wild: A Large-Scale Field Study." *ACM SIGMETRICS Performance Evaluation Review* 37(1). doi:10.1145/2492101.1555372.

In recent years, we've encountered some other zebras:

- Two web search requests hashed to the same 64-bit cache key, causing results for one request to be served in place of the other.
- C++ converted an `int64` to an `int` (only 32 bits), resulting in problems after 2^{32} requests (for more about this bug, see "Clean up code" on page 346).
- A bug in a distributed rebalancing algorithm was triggered only when the code ran simultaneously on hundreds of servers.
- Someone left a load test running for a week, causing performance to slowly degrade. We eventually determined that the machine was gradually suffering from memory allocation issues, leading to the degradation. We discovered this particular zebra because a normally short-lived test was left running for much longer than normal.
- Investigating slow C++ tests showed that the dynamic linker's loading time was superlinear in terms of the number of shared libraries loaded: at 10,000 shared libraries, it could take minutes to start running `main`.

When dealing with smaller, newer systems, expect horses (common bugs). When dealing with older, larger, and relatively stable systems, expect zebras (rare bugs)—operators have likely observed and fixed common bugs that surfaced over time. Issues are more likely to crop up in new parts of the system.

Data Corruption and Checksums

Memory may become corrupted for many reasons. Hardware problems, perhaps caused by environmental factors, are one cause. Software problems—for example, one thread writing while another reads—can also cause memory corruption. It is very dangerous for software engineers to think every weird bug is a hardware problem.

Modern DRAM provides several defenses against memory corruption:

- Error-correcting code (ECC) RAM corrects most corruptions.
- Detected, uncorrectable errors trigger a machine check exception, enabling the operating system to take action.
- An operating system can either panic or kill the process referencing corrupted memory. Either of these events prevents the system from using invalid memory contents.

Checksums are numbers derived from data for the purpose of detecting changes to the data. They can protect against unanticipated hardware and software failures. When using checksums, keep the following in mind:

- The tradeoff between the CPU cost of the checksum and the level of protection it provides
- Cyclic redundancy checks (CRCs) protect against single bit flips, while cryptographic hashes protect against human attackers. CRCs are much cheaper in terms of CPU time.
- The scope of the checksum

For example, imagine a client that sends a key and a value over the network to a server, which then stores the value on disk. A filesystem-level checksum only protects against filesystem or disk bugs. A checksum of the value generated by a client protects against network and server bugs. However, consider a bug where the client attempts to retrieve a value from the server. The client sends a key, but the buggy server reads the wrong value—and the checksum of the wrong value—from disk and returns those incorrect values to the client. Because the checksum's scope is limited to just the value, the checksum still passes, despite the bug. A better solution might include the key in both the value *and* the checksum. This solution would catch the case where the key and returned value do not match the checksum.

Set aside time for debugging and investigations

Both security investigations (to be discussed later) and debugging often take time—many hours of uninterrupted work. The temporary files scenario described in the previous section required somewhere between 5 and 10 hours of debugging. When running a major incident, give debuggers and investigators the space to focus by isolating them from the minute-by-minute response.

Debugging rewards slow, methodical, persistent approaches, in which people double-check their work and assumptions and are willing to dig deep. The temporary files problem also offers a negative example of debugging: the first responder initially diagnosed the outage as caused by user traffic and blamed poor system behavior on users. At the time, the team was in operational overload and experiencing pager fatigue due to nonurgent pages.

Chapter 17 of the SRE workbook discusses reducing operational overload. Chapter 11 of the SRE book suggests keeping ticket and pager volume below two per shift to give engineers time to dig deep into issues.

Record your observations and expectations

Write down what you see. Separately, write down your theories, even if you've already rejected them. Doing so has several advantages:

- It introduces structure to the investigation and helps you remember the steps you took during your investigation. When you start debugging, you don't know how long it will take to solve the issue—resolution might take five minutes or five months.
- Another debugger can read your notes, understand what you observed, and quickly participate in or take over the investigation. Your notes can help teammates avoid duplicative work, and may inspire others to think of new avenues for investigation. For more on this topic, see "Negative Results Are Magic" in Chapter 12 of the SRE book.
- In the case of potential security issues, it can be helpful to keep a log of each access and investigation step. Later, you may need to prove (sometimes in a court of law) which actions were performed by the attacker and which were performed by investigators.

After you've written down what you observed, write down what you expected to observe and why. Bugs often lurk in the space between your mental model of the system and its actual implementation. In the temporary files example, the developers assumed that deleting a file removed all references to it.

Know what's normal for your system

Often, debuggers start debugging what is actually an expected system behavior. Here are a few examples from our experience:

- A binary called `abort` near the end of its `shutdown` code. New developers saw the `abort` call in the logs and started debugging the call, not noticing that the interesting failure was actually the reason for the call to `shutdown`.
- When the Chrome web browser starts, it attempts to resolve three random domains (such as `cegzaukxwefark.local`) to determine whether the network is illicitly tampering with DNS. Even Google's own investigation team has mistaken these DNS resolutions for malware trying to resolve a command-and-control server hostname.

Debuggers often need to filter out these normal events, even if the events look relevant or suspicious. Security investigators have the added problem of a steady level of background noise and active adversaries that may be trying to hide their actions. You often need to filter out routine noisy activity like automated SSH login brute forcing, authentication errors caused by users' mistyped passwords, and port scanning before you can observe more serious issues.

One way to understand normal system behavior is to establish a baseline of system behavior when you don't suspect any problems. If you have a problem already, you

may be able to infer your baseline by examining historical logs from before the problem began.

For example, in Chapter 1 we described a global YouTube outage caused by a change to a generic logging library. The change caused the servers to run out of memory (OOM) and fail. Because the library was widely used within Google, our post-outage investigation questioned whether the outage had affected the number of OOMs for all other Borg tasks. While logs suggested that we had many OOM conditions that day, we were able to compare that data against a baseline of data from the previous two weeks, which showed that Google has many OOM conditions *every* day. Although the bug was serious, it did not meaningfully affect the OOM metric for Borg tasks.

Beware of normalizing deviance from best practices. Often, bugs become "normal behavior" over time, and you no longer notice them. For example, we once worked on a server that had spent ~10% of its memory in heap fragmentation. After many years of asserting that ~10% was the expected and therefore acceptable amount of loss, we examined a fragmentation profile and quickly found major opportunities for saving memory.

Operational overload and alert fatigue can lead you to grow a blind spot, and thus normalize deviance. To address normalized deviance, we actively listen to newcomers to the team, and to facilitate fresh perspectives, we rotate people in and out of on-call rotations and response teams—the process of writing documentation and explaining a system to others can also prompt you to question how well you understand a system. Additionally, we use Red Teams (see Chapter 20) to test our blind spots.

Reproduce the bug

If possible, attempt to reproduce the bug outside of the production environment. This approach has two main advantages:

- You don't impact systems serving actual users, so you can crash the system and corrupt data as much as you want.
- Because you don't expose any sensitive data, you can involve many people in the investigation without raising data security issues. You can also enable operations that aren't appropriate with actual user data, and capabilities like extra logging.

Sometimes, debugging outside of the production environment isn't feasible. Perhaps the bug triggers only at scale, or you can't isolate its trigger. The temporary files example is one such situation: we couldn't reproduce the bug with a full serving stack.

Isolate the problem

If you can reproduce the issue, the next step is to isolate the problem—ideally, to the smallest subset of code that still manifests it. You can do this by disabling components or temporarily commenting out subroutines until the problem is revealed.

In the temporary files example, once we observed that the memory management was acting strangely on all servers, we no longer had to debug all components on every affected machine. For another example, consider a single server (out of a large cluster of systems) that suddenly starts introducing high latency or errors. This scenario is the standard test of your monitoring, logs and other observability systems: can you quickly find a single bad server among the many servers in your system? See "What to Do When You're Stuck" on page 344 for more information.

You can also isolate problems within code. To provide a concrete example, we recently investigated memory usage for a program with a very limited memory budget. In particular, we examined the memory mappings for thread stacks. Although our mental model assumed that all threads had the same stack size, to our surprise we found that different thread stacks had many different sizes. Some stacks were quite large and risked consuming a big chunk of our memory budget. The initial debugging scope included the kernel, glibc, Google's threading library, and all code that started threads. A trivial example based around glibc's `pthread_create` created thread stacks of the same size, so we could rule out the kernel and glibc as the sources of the different sizes. We then examined the code that started threads, and discovered that many libraries just picked a thread size at random, explaining the variation of sizes. This understanding enabled us to save memory by focusing on the few threads with large stacks.

Be mindful of correlation versus causation

Sometimes debuggers assume that two events that start at the same time, or that exhibit similar symptoms, have the same root cause. However, correlation does not always imply causation. Two mundane problems might occur at the same time but have different root causes.

Some correlations are trivial. For example, an increase in latency might lead to a reduction in user requests, simply because users are waiting longer for the system to respond. If a team repeatedly discovers correlations that in retrospect are trivial, there might be a gap in their understanding of how the system is supposed to work. In the temporary files example, if you know that the failure to delete files results in full disks, you won't be surprised by the correlation.

However, our experience has shown that investigating correlations is often useful—notably, correlations that occur at the start of outages. You can home in on likely causes by thinking, "*X* is broken, *Y* is broken, *Z* is broken; what's the common element among the three?" We've also had some success with correlation-based tooling.

For example, we deployed a system that automatically correlates machine problems with the Borg tasks running on the machine. As a result, we can often identify a suspicious Borg task causing a widespread problem. This kind of automated tooling produces much more effective, statistically stronger, and faster correlations than human observation.

Errors can also manifest during deployment—see Chapter 12 in the SRE book. In simple situations, the new code being deployed may have problems, but deployments can also trigger latent bugs in old systems. In these situations, debugging may erroneously focus on the new code being deployed, rather than the latent issues. Systematic investigation—determining what is happening, and why—helps in these cases. In one example we witnessed, the old code had much worse performance than the new code, which resulted in an accidental throttle on the system as a whole. When its performance improved, other parts of the system became overloaded instead. The outage was correlated with the new deployment, but the deployment was not the root cause.

Test your hypotheses with actual data

When debugging, it can be tempting to speculate about the root causes of issues before actually looking at the system. When it comes to performance issues, this tendency introduces blind spots, as the problems often lie in code the debuggers have not looked at in a long time. As an example, once we were debugging a web server that was running slowly. We assumed the problem lay in the backends, but a profiler showed that the practice of logging every possible scrap of input to disk and then calling `sync` was causing vast amounts of delay. We discovered this only when we set aside our initial assumptions and dug into the system more deeply.

Observability is the property of being able to determine what your system is doing by examining its outputs. Tracing solutions like Dapper and Zipkin are very useful for this kind of debugging. Debugging sessions start with basic questions like, "Can you find a slow Dapper trace?"[6]

It can be challenging for beginners to determine what tool is best for the job, or even what tools exist. Brendan Gregg's *Systems Performance* (Prentice Hall, 2013), which provides an exhaustive tour of tooling and techniques, is a fantastic reference for performance debugging.

6 These tools often require some setup; we will discuss them further in "What to Do When You're Stuck" on page 344.

Reread the docs

Consider the following guidance from the Python documentation:

> There are no implied relationships among the comparison operators. The truth of `x==y` does not imply that `x!=y` is false. Accordingly, when defining `__eq__()`, one should also define `__ne__()` so that the operators will behave as expected.

Recently, a Google team spent a good amount of time debugging an internal dashboard optimization. When they got stuck, the team reread the documentation and discovered a plainly written warning message that explained why the optimization had never worked at all. People were so used to the dashboard's slow performance, they failed to notice that the optimization was completely ineffective.[7] Initially, the bug seemed remarkable; the team thought it was an issue in Python itself. After they found the warning message, they determined that it wasn't a zebra, it was a horse—their code had never worked.

Practice!

Debugging skills stay fresh only if you use them often. You can speed up your investigations and keep the tips we've provided here fresh in your mind by staying familiar with relevant tools and logs. Regularly practicing debugging also provides the opportunity to script the common and tedious parts of the process—for example, automation to examine logs. To get better at debugging (or to stay sharp), practice, and keep the code you write during debugging sessions.

At Google, we formally practice debugging with regular large-scale disaster recovery tests (referred to as DiRT, or the Disaster Recovery Testing program)[8] and security penetration tests (see Chapter 16). Smaller-scale tests, involving one or two engineers in a room for an hour, are much easier to set up and are still quite valuable.

What to Do When You're Stuck

What should you do when you've been investigating an issue for days and still have no idea what caused the problem? Maybe it manifests only in the production environment, and you can't reproduce the bug without affecting live users. Maybe while mitigating the problem, you lost important debugging information when the logs rotated. Maybe the nature of the problem prevents useful logging. We once debugged an issue where a memory container ran out of RAM and the kernel issued a SIGKILL for all processes in the container, stopping all logging. Without logs, we couldn't debug the issue.

7 This is another example of normalized deviance, where people get used to suboptimal behavior!

8 See Krishnan, Kripa. 2012. "Weathering the Unexpected." *ACM Queue* 10(9). *https://oreil.ly/xFPfT*.

A key strategy in these situations is to improve the debugging process. Sometimes, using the methods for developing postmortems (as described in Chapter 18) may suggest ways forward. Many systems are in production for years or decades, so efforts to improve debugging are nearly always worthwhile. This section describes some approaches to improving your debugging methods.

Improve observability

Sometimes you need to see what a bit of code is doing. Is this code branch used? Is this function used? Could this data structure be large? Is this backend slow at the 99th percentile? What backends is this query even using? In these situations, you need better visibility into the system.

In some cases, methods like adding more structured logging to improve observability are straightforward. We once investigated a system that monitoring showed was serving too many 404 errors,[9] but the web server wasn't logging these errors. After adding additional logging for the web server, we discovered that malware was attempting to fetch erroneous files from the system.

Other debugging improvements take serious engineering effort. For example, debugging a complex system like Bigtable requires sophisticated instrumentation. The Bigtable master is the central coordinator for a Bigtable zone. It stores the list of servers and tablets in RAM, and several mutexes protect these critical sections. As Bigtable deployments at Google grew over time, the Bigtable master and these mutexes became a scaling bottleneck. To get more visibility into possible problems, we implemented a wrapper around a mutex that exposes stats such as queue depth and the time the mutex is held.

Tracing solutions like Dapper and Zipkin are very useful for this kind of complex debugging. For example, suppose you have a tree of RPCs, with the frontend calling a server, which calls another server, and so on. Each RPC tree has a unique ID assigned at the root. Each server then logs traces about the RPCs it receives, sends, and so on. Dapper collects all traces centrally and joins them via ID. This way, a debugger can see all the backends touched by the user's request. We've found Dapper to be critical to understanding latency in distributed systems. Similarly, Google embeds a simple web server in nearly every binary to provide visibility into each binary's behavior. The server has debugging endpoints that provide counters, a symbolized dump of all the running threads, inflight RPCs, and so on. For more information, see Henderson (2017).[10]

9 404 is a standard HTTP error code for "file not found."

10 Henderson, Fergus. 2017. "Software Engineering at Google." arXiv preprint arXiv:1702.01715v2.

Observability is not a substitute for understanding your system. Nor is it a substitute for critical thinking when debugging (sadly!). Often, we've found ourselves frantically adding more logging and counters in an effort to see what the system is doing, but what's really going on becomes clear only when we've taken a step back and thought about the problem.

Observability is a large and quickly evolving topic, and it's useful for more than debugging.[11] If you're a smaller organization with limited developer resources, you can consider using open source systems or purchasing a third-party observability solution.

Take a break

Giving yourself a bit of distance from an issue can often lead to new insights when you return to the problem. If you've been working heads-down on debugging and hit a lull, take a break: drink some water, go outside, get some exercise, or read a book. Bugs sometimes make themselves evident after a good sleep. A senior engineer in our forensic investigation team keeps a cello in the team's lab. When he's truly stuck on a problem, it's common for him to retreat into the lab for 20 minutes or so to play; he then comes back reenergized and refocused. Another investigator keeps a guitar handy, and others keep sketching and doodling pads in their desks so they can draw or create a silly animated GIF to share with the team when they need that mental realignment.

Make sure to also maintain good team communication. When you step aside to take a break, let the team know that you need a recharge and are following best practices. It's also helpful to document where you are in the investigation and why you're stuck. Doing so makes it easier for another investigator to pick up your work, and for you to return to the place where you left off. Chapter 17 has more advice on maintaining morale.

Clean up code

Sometimes you suspect there's a bug in a chunk of your code, but can't see it. Trying to generically improve code quality may help in this situation. As mentioned earlier in this chapter, we once debugged a bit of code that failed in production after 2^{32} requests because C++ was converting an `int64` into an `int` (only 32 bits) and truncating it. Although the compiler can warn you about such conversions using `-Wconversion`, we weren't using the warning because our code had many benign conversions. Cleaning up the code enabled us to use the compiler warning to detect more possible bugs and prevent new bugs related to conversion.

11 For a comprehensive survey of the topic, see Cindy Sridharan's "Monitoring in the Time of Cloud Native" blog post.

Here are some other tips for cleanup:

- Improve unit test coverage. Target functions where you suspect bugs may lie, or that have a track record of being buggy. (See Chapter 13 for more information.)
- For concurrent programs, use sanitizers (see "Sanitize Your Code" on page 267) and annotate mutexes.
- Improve error handling. Often, adding some more context around an error is sufficient to expose the problem.

Delete it!

Sometimes bugs lurk in legacy systems, especially if developers haven't had time to get or stay familiar with the codebase or maintenance has lapsed. A legacy system might also be compromised or incur new bugs. Instead of debugging or hardening the legacy system, consider deleting it.

Deleting a legacy system can also improve your security posture. For example, one of the authors was once contacted (through Google's Vulnerability Reward Program, as described in Chapter 20) by a security researcher who had found a security issue in one of our team's legacy systems. The team had previously isolated this system to its own network, but hadn't upgraded the system in quite some time. Newer members of the team were unaware that the legacy system even existed. To address the researcher's discovery, we decided to remove the system. We no longer needed most of the functionality it provided, and we were able to replace it with a considerably simpler modern system.

Be thoughtful when rewriting legacy systems. Ask yourself why your rewritten system will do a better job than the legacy system. Sometimes, you might want to rewrite a system because it's fun to add new code, and debugging old code is tedious. There are better reasons for replacing systems: sometimes requirements for the system change, and with a small amount of work, you can remove the old system. Alternatively, perhaps you've learned something from the first system and can incorporate this knowledge to make the second system better.

Stop when things start to go wrong

Many bugs are tricky to find because the source and its effects can be far apart in the system. We recently encountered an issue where network gear was corrupting internal DNS responses for hundreds of machines. For example, programs would perform a DNS lookup for the machine `exa1`, but receive the address of `exa2`. Two of our systems had different responses to this bug:

- One system, an archival service, would connect to `exa2`, the wrong machine. However, the system then checked that the machine to which it had connected was the expected machine. Since the machine names didn't match, the archival service job failed.
- Another system that collected machine metrics would collect metrics from the wrong machine, `exa2`. The system then triggered repairs on `exa1`. We detected this behavior only when a technician pointed out that they'd been asked to repair the fifth disk of a machine that didn't have five disks.

Of these two responses, we prefer the behavior of the archival service. When issues and their effects are far apart in the system—for example, when the network is causing application-level errors—having applications fail closed can prevent downstream effects (such as suspecting disk failure on the wrong system). We cover the topic of whether to fail open or fail closed in greater depth in Chapter 8.

Improve access and authorization controls, even for nonsensitive systems

It's possible to have "too many cooks in the kitchen"—that is, you may run into debugging situations where many people could be the source of a bug, making it difficult to isolate the cause. We once responded to an outage caused by a corrupted database row, and we couldn't locate the source of the corrupt data. To eliminate the possibility that someone could write to the production database by mistake, we minimized the number of roles that had access and required a justification for any human access. Even though the data was not sensitive, implementing a standard security system helped us prevent and investigate future bugs. Thankfully, we were also able to restore that database row from backup.

Collaborative Debugging: A Way to Teach

Many engineering teams teach debugging by working collectively on actual live issues in person (or over a videoconference). In addition to keeping experienced debuggers' skills fresh, collaborative debugging helps to build psychological safety for new team members: they have the opportunity to see the best debuggers on the team get stuck, backtrack, or otherwise struggle, which shows them that it's OK to be wrong and to have a hard time.[12] For more on security education, see Chapter 21.

12 See Julia Rozovsky's blog post "The Five Keys to a Successful Google Team" and the *New York Times* article "What Google Learned from Its Quest to Build the Perfect Team" by Charles Duhigg.

We've found that the following rules optimize the learning experience:

- Only two people should have laptops open:
  - A "driver," who performs actions requested by the others
  - A "note taker"
- Every action should be determined by the audience. Only the driver and the note taker are permitted to use computers, but they do not determine the actions taken. This way, participants don't perform solo debugging, only to present an answer without sharing their thought processes and troubleshooting steps.

The team collectively identifies one or more problems to examine, but no one in the room should know in advance how to solve the issues. Each person can request that the driver perform an action to troubleshoot the issue (for example, open a dashboard, look at logs, reboot a server, etc.). Since everyone is present to witness the suggestions, everyone can learn about tools and techniques that participants suggest. Even very experienced team members learn new things from these exercises.

As described in Chapter 28 of the SRE book, some teams also use "Wheel of Misfortune" simulation exercises. These exercises can either be theoretical, with verbal walk-throughs of problem solving, or practical, where the test giver induces a fault in a system. These scenarios also involve two roles:

- The "test giver," who constructs and presents the test
- The "test taker," who attempts to solve the problem, perhaps with the help of their teammates

Some teams prefer the safe environment of staged exercises, but practical Wheel of Misfortune exercises require nontrivial setup, whereas most systems always have a live issue to collectively debug. Regardless of the approach, it's important to maintain an inclusive learning environment where everyone feels safe to actively contribute.

Both collaborative debugging and Wheel of Misfortune exercises are excellent ways to introduce new techniques to your team and reinforce best practices. People can see how the techniques are useful in real-world situations, often for the trickiest of problems. Teams also get some practice debugging issues together, making them more effective when a real crisis occurs.

How Security Investigations and Debugging Differ

We expect every engineer to debug systems, but we advise that trained and experienced security and forensic specialists investigate system compromises. When the line between "bug investigation" and "security problem" is unclear, there's an opportunity for collaboration between the two sets of specialist teams.

A *bug investigation* often begins when a system experiences a problem. The investigation focuses on what happened in the system: what data was sent, what happened with that data, and how the service began acting contrary to its intent. *Security investigations* begin a little differently, and quickly pivot to questions like: What has the user who submitted that job been doing? What other activity is that user responsible for? Do we have a live attacker in the system? What will the attacker do next? In short, debugging is more code-focused, while a security investigation may quickly focus on the adversary behind an attack.

The steps we recommended previously for debugging issues may also be counterproductive during security investigations. Adding new code, deprecating systems, and so on may have unintended side effects. We've responded to a number of incidents where a debugger removed files that normally didn't belong on the system in the hopes of resolving errant behavior, and it turned out that those files had been introduced by the attacker, who was thus alerted to the investigation. In one case, the attacker even responded in kind by deleting the entire system!

Once you suspect that a security compromise has occurred, your investigation may also take on a new sense of urgency. The possibility that a system is being intentionally subverted raises questions that feel serious and pressing. What is the adversary after? What other systems may be subverted? Do you need to call law enforcement or regulators? Security investigations grow organically in complexity as the organization begins to address operational security concerns (Chapter 17 discusses this topic further). Experts from other teams, such as Legal, may get involved before you can begin your investigation. In short, the moment you suspect a security compromise is a good time to get help from security professionals.

Intersection of Security and Reliability: Recognizing a Compromise

During an investigation, you may come across many inconsistencies that are simply oddities in a system that should be corrected. However, if these inconsistencies start to add up, or you encounter an increasing number of inconsistencies that have no explanation, an adversary might be tampering with your systems. Here's a sample list of inconsistencies that might raise suspicions about a potential compromise:

- Logs and data necessary for debugging are missing, truncated, or corrupted.
- You began your investigation looking at system behavior, but find yourself developing your hypothesis around the actions of *an account* or *a user*.
- The system is behaving in ways that you can't explain as accidental. For example, your web server keeps spawning interactive shells when it appears to crash.
- Files have characteristics that seem abnormal for the file type (e.g., you find a file called *rick_roll.gif*, but it is 1 GB in size and appears to have ZIP or TAR headers).
- Files appear to have been intentionally hidden, misnamed in familiar ways (e.g., *explore.exe* on Windows), or otherwise obfuscated.

Deciding when to stop investigating and declare a security incident can be a difficult judgment call. Many engineers have a natural inclination to refrain from "making a scene" by escalating issues that aren't yet proven to be security-related, but continuing the investigation to the point of proof may be the wrong move. Our advice is to remember "horses versus zebras": the vast majority of bugs are, in fact, bugs, not malicious actions. However, *also* keep a vigilant eye open for those black and white stripes zipping by.

Collect Appropriate and Useful Logs

At their heart, logs and system crash dumps are both just information you can collect to help you understand what happened in a system and to investigate problems—both accidental and intentional.

Before you launch any service, it's important to consider the kinds of data the service will store on behalf of users, and the pathways to access the data. Assume that any action that leads to data or system access may be in scope for a future investigation, and that someone will need to audit that action. Investigating any service issue *or* security issue depends heavily on logs.

Our discussion here of "logs" refers to structured, timestamped records from systems. During investigations, analysts may also rely heavily on other sources of data, like core dumps, memory dumps, or stack traces. We recommend handling those systems as much like logs as possible. Structured logs are useful for many different business purposes, such as per-usage billing. However, we focus here on structured logs collected for security investigations—the information you need to collect now so it's available in the event of a future issue.

Design Your Logging to Be Immutable

The system you build to collect logs should be immutable. When log entries are written, it should be difficult to alter them (but not impossible; see "Take Privacy into Consideration" on page 352), and alterations should have an immutable audit trail. Attackers commonly erase traces of their activity on a system from all log sources as soon as they establish a solid foothold. A common best practice to counter this tactic is to write your logs remotely to a centralized and distributed log server. This increases the attacker's workload: in addition to compromising the original system, they also have to compromise the remote log server. Be sure to harden the log system carefully.

Before the age of modern computing, extra-critical servers logged directly to an attached line printer, like the one in Figure 15-1, which printed log records to paper as they were generated. In order to erase their traces, a remote attacker would have needed someone to physically remove paper from the printer and burn it!

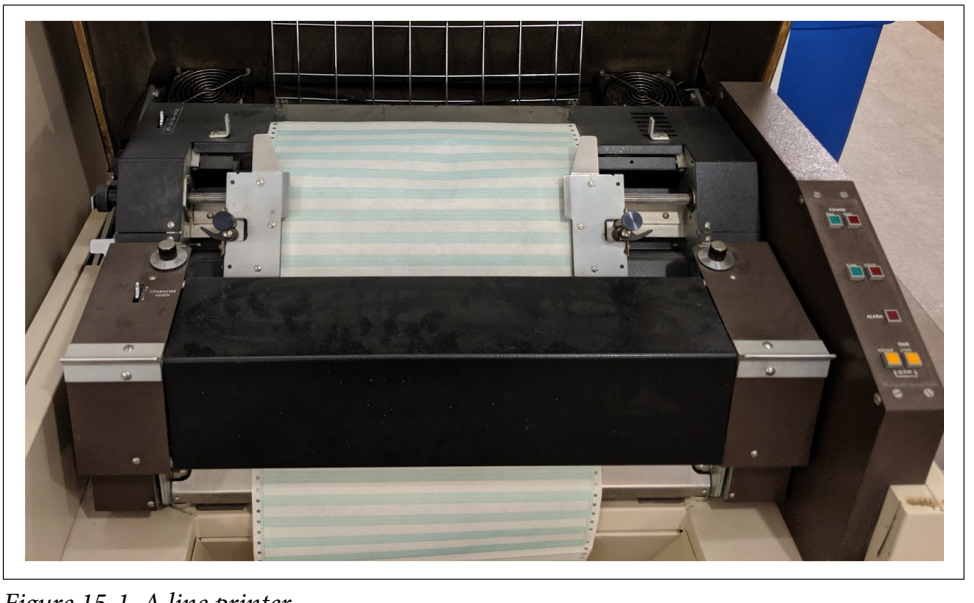

Figure 15-1. A line printer

Take Privacy into Consideration

The need for privacy-preserving features is an increasingly important factor in the design of systems. While privacy is not a focus of this book, you will likely need to take into account local regulations and your organization's privacy policies when designing logging for security investigations and debugging. Be sure to consult with any privacy and legal colleagues in your organization on this topic. Here are some topics you may want to discuss:

Depth of logging

To be maximally useful for any investigation, logs need to be as complete as possible. A security investigation might need to examine every action a user (or an attacker using their account) performed inside a system, the host from which they logged in, and the exact times at which the events occurred. Agree on an organizational policy about what information is acceptable to log, given that many privacy-preserving techniques discourage retaining sensitive user data in logs.

Retention

For some investigations, it can be beneficial to retain logs for a long time. According to a 2018 study, it takes most organizations an average of about 200 days to discover a system compromise. Insider threat investigations at Google have relied on operating system security logs going back several years. How long you can keep logs is an important discussion to have within your organization.

Access and audit controls

Many of the controls we recommend for protecting data also apply to logs. Be sure to protect logs and metadata just as you protect other data. See Chapter 5 for relevant strategies.

Data anonymization or pseudonymization

Anonymizing unnecessary data components—either as they're written or after a period of time—is one increasingly common privacy-preserving method for handling logs. You can even implement this functionality such that investigators and debuggers can't determine who a given user is, but can clearly build a timeline of that user's actions throughout their session for debugging purposes. Anonymization is tricky to get right. We recommend consulting privacy specialists and reading published literature on this topic.[13]

Encryption

You can also implement privacy-preserving logging using asymmetric encryption of data. This encryption method is ideal for protecting log data: it uses a nonsensitive "public key" that anyone can use to write data securely, but requires a secret (private) key to decrypt the data. Design options like daily key pairs can allow debuggers to obtain small subsets of log data from recent system activity, while preventing someone from obtaining large amounts of log data in aggregate. Be sure to carefully consider how you store keys.

13 See, e.g., Ghiasvand, Siavash, and Florina M. Ciorba. 2017. "Anonymization of System Logs for Privacy and Storage Benefits." arXiv preprint arXiv:1706.04337. See also Jan Lindquist's article on pseudonymization of personal data for General Data Protection Regulation (GDPR) compliance.

Determine Which Security Logs to Retain

Although security engineers would often prefer having too many logs to having too few, it pays to be a bit selective in what you log and retain. Storing an overabundance of logs can be costly (as discussed in "Budget for Logging" on page 357), and sifting through excessively large data sets can slow down an investigator and use a large amount of resources. In this section, we discuss some types of logs you may want to capture and retain.

Operating system logs

Most modern operating systems have built-in logging. Windows has Windows Event logs, while Linux and Mac have syslog and auditd logs. Many vendor-supplied appliances (such as camera systems, environmental controls, and fire alarm panels) have a standard operating system that also produces logs (such as Linux) under the hood. Built-in logging frameworks are useful for investigations, and using them requires almost no effort because they're often enabled by default or easily configured. Some mechanisms, like auditd, are not enabled by default for performance reasons, but enabling them can be an acceptable tradeoff in real-world use.

Host agents

Many companies choose to enable additional logging capabilities by installing a *host intrusion detection system* (HIDS) or *host agent* on workstations and servers.

Antivirus Software

Antivirus software scans files for patterns that indicate known malware (for example, a code sequence unique to a particular virus) and looks for suspicious behavior (for example, attempts to modify sensitive system files). Experts don't always agree on the value and use of antivirus software in a modern security setting.[14] We think that having antivirus protection deployed on every endpoint computer has become less useful over time as threats have grown more sophisticated. Additionally, if poorly authored, antivirus software can even introduce more security risk to the system.

Modern (sometimes referred to as "next-gen") host agents use innovative techniques aimed at detecting increasingly sophisticated threats. Some agents blend system and user behavior modeling, machine learning, and threat intelligence to identify previously unknown attacks. Other agents are more focused on gathering additional data

14 See, e.g., Joxean Koret's presentation "Breaking Antivirus Software" at 44CON 2014.

about the system's operation, which can be useful for offline detection and debugging activities. Some, such as OSQuery and GRR, provide real-time visibility into a system.

Host agents always impact performance, and are often a source of friction between end users and IT teams. Generally speaking, the more data an agent can gather, the greater its performance impact may be because of deeper platform integration and more on-host processing. Some agents run as part of the kernel, while others run as userspace applications. Kernel agents have more functionality, and are therefore typically more effective, but they can suffer from reliability and performance issues as they try to keep up with operating system functionality changes. Agents that run as applications are much easier to install and configure, and tend to have fewer compatibility problems. The value and performance of host agents varies widely, so we recommend thoroughly evaluating a host agent before using it.

Application logs

Logging applications—whether vendor-supplied like SAP and Microsoft SharePoint, open source, or custom-written—generate logs that you can collect and analyze. You can then use these logs for custom detection and to augment investigation data. For example, we use application logs from Google Drive to determine if a compromised computer has downloaded sensitive data.

When developing custom applications, collaboration between security specialists and developers can ensure that the applications log security-relevant actions, such as data writes, changes in ownership or state, and account-related activity. As we mention in "Improve observability" on page 345, instrumenting your applications for logging can also facilitate debugging for esoteric security and reliability issues that would otherwise be difficult to triage.

Cloud logs

Increasingly, organizations are moving parts of their business or IT processes to cloud-based services, ranging from data in Software-as-a-Service (SaaS) applications to virtual machines running critical customer-facing workloads. All of these services present unique attack surfaces and generate unique logs. For example, an attacker can compromise the account credentials for a cloud project, deploy new containers to the project's Kubernetes cluster, and use those containers to steal data from the cluster's accessible storage buckets. Cloud computing models commonly launch new instances daily, which makes detecting threats in the cloud dynamic and complex.

When it comes to detecting suspicious activity, cloud services present advantages and disadvantages. Using services like Google's BigQuery, it's easy and relatively cheap to collect and store large amounts of log data, and even to run detection rules, directly in the cloud. Google Cloud services also offer built-in logging solutions like Cloud Audit Logs and Stackdriver Logging. On the other hand, because there are many

kinds of cloud services, it can be hard to identify, enable, and centralize all the logs you need. Because it's easy for developers to create new IT assets in the cloud, many companies find it difficult to identify all of their cloud-based assets. Cloud service providers may also predetermine which logs are available to you, and these options may not be configurable. It's important to understand the limitations of your provider's logging and your potential blind spots.

Taking Inventory of Your Assets

While building the pipelines you need to investigate system problems, consider building a set of tools to use that data to identify your cloud assets, including the assets you might not know about. To provide an example for the latter case, integrating with your financial system to look for bills being paid to cloud providers may reveal system components that need further attention. In an article about Google's BeyondCorp architecture, we referenced a set of tools called the Device Inventory Service, which we leverage to discover assets that may belong to Google (or contain Google's data). Building a similar service and extending it to your cloud environments is one way to identify these types of assets.

A variety of commercial software, often itself based in the cloud, aims to detect attacks against cloud services. Most of the established cloud providers offer integrated threat detection services, such as Google's Event Threat Detection. Many companies combine these built-in services with internally developed detection rules or third-party products.

Cloud access security brokers (CASBs) are a notable category of detection and prevention technology. CASBs function as intermediaries between end users and cloud services to enforce security controls and provide logging. For example, a CASB might prevent your users from uploading certain kinds of files, or log every file downloaded by a user. Many CASBs have a detection function that alerts the detection team about potentially malicious access. You can also integrate logs from the CASB into custom detection rules.

Network-based logging and detection

Since the late 1990s, the use of *network intrusion detection systems* (NIDSs) and *intrusion prevention systems* (IPSs) that capture and inspect network packets has been a common detection and logging technique. IPSs also block some attacks. For example, they may capture information about which IP addresses have exchanged traffic, along with limited information about that traffic, such as packet size. Some IPSs may have the ability to record the entire contents of certain packets, based on customizable criteria—for example, packets sent to high-risk systems. Others can also detect malicious activity in real time and send alerts to the appropriate team. Since these systems

are very useful and have few downsides beyond cost, we highly recommend them for almost any organization. However, think carefully about who can effectively triage the alerts they generate.

Logs of DNS queries are also useful network-based sources. DNS logs enable you to see whether any computer at the company has resolved a hostname. For example, you might want to see whether any host on your network has performed a DNS query for a known malicious hostname, or you may want to examine previously resolved domains to identify every machine visited by an attacker that had control of your system. A security operations team might also use DNS "sinkholes" that falsely resolve known malicious domains so they can't be effectively used by attackers. Detection systems then tend to trigger a high-priority alert when users access a sinkholed domain.

You can also use logs from any web proxies used for internal or egress traffic. For example, you can use a web proxy to scan web pages for indicators of phishing or known vulnerability patterns. When using a proxy for detection, you'll also want to consider employee privacy and discuss the use of proxy logs with a legal team. Generally speaking, we recommend tailoring your detection as closely as possible to malicious content in order to minimize the amount of employee data you encounter while triaging alerts.

Budget for Logging

Debugging and investigative activities use resources. One system we worked on had 100 TB of logs, which were mostly never used. Because logging consumes a significant amount of resources, and logs are often monitored less frequently in the absence of problems, it can be tempting to underinvest in the logging and debugging infrastructure. To avoid this, we strongly recommend that you budget for logging in advance, taking into account how much data you may need to resolve a service issue or security incident.

Modern log systems often incorporate a relational data system (e.g., Elasticsearch or BigQuery) to quickly and easily query data in real time. The cost of this system grows along with the number of events it needs to store and index, the number of machines it needs to process and query the data, and the storage space required. When retaining data for long periods of time, it's therefore useful to prioritize logs from relevant data sources for longer-term storage. This is an important tradeoff decision: if an attacker is good at hiding their tracks, it may take you quite some time to discover that an incident has occurred. If you store only a week's worth of access logs, you may not be able to investigate an intrusion at all!

Intersection of Security and Reliability: Budgeting for Long-Term Log Storage

Your need to retain log data for long periods of time may come into conflict with your budget. What if you can't afford enough log storage and processing infrastructure to keep as many logs as you'd like? To balance the costs of storage against your retention needs, we recommend designing your logs to *degrade gracefully*, instead of halting log collection when you reach storage capacity.

Data summarization is an increasingly common method to implement graceful degradation. For example, you might build a log system that dedicates 90% of storage to full log retention, and reserves the remaining 10% for lower-fidelity summarizations of some data. To avoid having to reread and parse already written logs (which is computationally expensive), your system could write two log files at collection time: the first log contains full-fidelity data, while the second contains a summarization. As your storage reaches capacity, you can delete the larger logs to free up storage space, and retain the smaller files as longer-lived data. For example, you can produce both full packet captures and netflow data, deleting the captures after *N* days but keeping the netflow data for a year. The logs will then have much lower storage costs, but still provide key intelligence about the information that hosts communicate.

We also recommend the following investment strategies for security-focused log collection:

- Focus on logs that have a good signal-to-noise ratio. For example, firewalls routinely block many packets, most of which are harmless. Even malicious packets blocked by a firewall may not be worth paying attention to. Gathering logs for these blocked packets could use a tremendous amount of bandwidth and storage for almost no benefit.
- Compress logs whenever possible. Because most logs contain a lot of duplicated metadata, compression is typically very effective.
- Separate storage into "warm" and "cold." You can offload logs from the distant past to cheap offline cloud storage ("cold storage"), while retaining logs that are more recent or related to known incidents on local servers for immediate use ("warm storage"). Similarly, you might store compressed raw logs for a long time, but put only recent logs in the expensive relational database with full indexing.
- Rotate logs intelligently. Generally, it's best to delete the oldest logs first, but you may want to retain the most important log types longer.

Robust, Secure Debugging Access

To debug issues, you often need access to the systems and the data they store. Can a malicious or compromised debugger see sensitive information? Can a failure of a security system (and remember: all systems fail!) be resolved? You need to ensure that your debugging systems are reliable and secure.

Reliability

Logging is another way systems can fail. For example, a system can run out of disk space to store logs. Failing open in this example entails another tradeoff: the approach can make your entire system more resilient, but an attacker can potentially disrupt your logging mechanism.

Plan for situations where you might need to debug or repair the security systems themselves. Consider the tradeoffs necessary to make sure you don't lock yourself out of a system, but can still keep it secure. In this case, you might consider keeping a set of emergency-only credentials, offline in a secure location, that set off high-confidence alarms when used. As an example, a recent Google network outage caused high packet loss. When responders attempted to obtain internal credentials, the authentication system could not reach one backend and failed closed. However, emergency credentials enabled the responders to authenticate and fix the network.

Security

One system we worked on, used for phone support, allowed administrators to impersonate a user and to view the UI from their perspective. As a debugger, this system was wonderful; you could clearly and quickly reproduce a user's problem. However, this type of system provides possibilities for abuse. Debugging endpoints—from impersonation to raw database access—need to be secured.

For many incidents, debugging unusual system behavior need not require access to user data. For example, when diagnosing TCP traffic problems, the speed and quality of bytes on the wire is often enough to diagnose issues. Encrypting data in transit can protect it from any possible attempt by third parties to observe it. This has the fortunate side effect of allowing more engineers access to packet dumps when needed. However, one possible mistake is to treat metadata as nonsensitive. A malicious actor can still learn a lot about a user from metadata by tracking correlated access patterns—for instance, by noting the same user accessing a divorce lawyer and a dating site in the same session. You should carefully assess the risks from treating metadata as nonsensitive.

Also, some analysis *does* require actual data—for example, finding frequently accessed records in a database, and then figuring out why these accesses are common. We once debugged a low-level storage problem caused by a single account receiving

thousands of emails per hour. "Zero Trust Networking" on page 62 has more information about access control for these situations.

Conclusion

Debugging and investigations are necessary aspects of managing a system. To reiterate the key points in the chapter:

- *Debugging* is an essential activity whereby systematic techniques—not guesswork—achieve results. You can make debugging vastly easier by implementing tools or logging to provide visibility into the system. Practice debugging to hone your skills.
- *Security investigations* are different from debugging. They involve different people, tactics, and risks. Your investigation team should include experienced security professionals.
- *Centralized logging* is useful for debugging purposes, critical for investigations, and often useful for business analysis.
- *Iterate* by looking at some recent investigations and asking yourself what information would have helped you debug an issue or investigate a concern. Debugging is a process of continuous improvement; you will regularly add data sources and look for ways to improve observability.
- *Design for safety*. You need logs. Debuggers need access to systems and stored data. However, as the amount of data you store increases, both logs and debugging endpoints can become targets for adversaries. Design logging systems to collect information you'll need, but also to require robust permissions, privileges, and policies to obtain that data.

Both debugging and security investigations often depend on sudden insight and luck, and even the best debuggers are sometimes sadly left in the dark. Remember that chance favors the prepared: by being ready with logs, and a system for indexing and investigating them, you can take advantage of the chances that come your way. Good luck!