CHAPTER 1

The Intersection of Security and Reliability

By Adam Stubblefield, Massimiliano Poletto, and Piotr Lewandowski with David Huska and Betsy Beyer

On Passwords and Power Drills

On September 27, 2012, an innocent Google-wide announcement caused a series of cascading failures in an internal service. Ultimately, recovering from these failures required a power drill.

Google has an internal password manager that allows employees to store and share secrets for third-party services that don't support better authentication mechanisms. One such secret is the password to the guest WiFi system on the large fleet of buses that connect Google's San Francisco Bay Area campuses.

On that day in September, the corporate transportation team emailed an announcement to thousands of employees that the WiFi password had changed. The resulting spike in traffic was far larger than the password management system—which had been developed years earlier for a small audience of system administrators—could handle.

The load caused the primary replica of the password manager to become unresponsive, so the load balancer diverted traffic to the secondary replica, which promptly failed in the same way. At this point, the system paged the on-call engineer. The engineer had no experience responding to failures of the service: the password manager was supported on a best-effort basis, and had never suffered an outage in its five years of existence. The engineer attempted to restart the service, but did not know that a restart required a hardware security module (HSM) smart card.

These smart cards were stored in multiple safes in different Google offices across the globe, but not in New York City, where the on-call engineer was located. When the service failed to restart, the engineer contacted a colleague in Australia to retrieve a smart card. To their great dismay, the engineer in Australia could not open the safe because the combination was stored in the now-offline password manager. Fortunately, another colleague in California had memorized the combination to the on-site safe and was able to retrieve a smart card. However, even after the engineer in California inserted the card into a reader, the service still failed to restart with the cryptic error, "The password could not load any of the cards protecting this key."

At this point, the engineers in Australia decided that a brute-force approach to their safe problem was warranted and applied a power drill to the task. An hour later, the safe was open—but even the newly retrieved cards triggered the same error message.

It took an additional hour for the team to realize that the green light on the smart card reader did not, in fact, indicate that the card had been inserted correctly. When the engineers flipped the card over, the service restarted and the outage ended.

Reliability and security are both crucial components of a truly trustworthy system, but building systems that are both reliable and secure is difficult. While the requirements for reliability and security share many common properties, they also require different design considerations. It is easy to miss the subtle interplay between reliability and security that can cause unexpected outcomes. The password manager's failure was triggered by a reliability problem—poor load-balancing and load-shedding strategies—and its recovery was later complicated by multiple measures designed to increase the security of the system.

The Intersection of Security and Privacy

Security and privacy are closely related concepts. In order for a system to respect user privacy, it must be fundamentally secure and behave as intended in the presence of an adversary. Similarly, a perfectly secure system doesn't meet the needs of many users if it doesn't respect user privacy. While this book focuses on security, you can often apply the general approaches we describe to achieve privacy objectives, as well.

Reliability Versus Security: Design Considerations

In designing for reliability and security, you must consider different risks. The primary reliability risks are nonmalicious in nature—for example, a bad software update or a physical device failure. Security risks, however, come from adversaries who are actively trying to exploit system vulnerabilities. When designing for reliability, you assume that some things will go wrong at some point. When designing for security,

you must assume that an adversary could be trying to make things go wrong at any point.

As a result, different systems are designed to respond to failures in quite different ways. In the absence of an adversary, systems often fail *safe* (or *open*): for example, an electronic lock is designed to remain open in case of power failure, to allow safe exit through the door. Fail safe/open behavior can lead to obvious security vulnerabilities. To defend against an adversary who might exploit a power failure, you could design the door to fail *secure* and remain closed when not powered.

Reliability and Security Tradeoff: Redundancy

In designing for reliability, you often need to add redundancy to systems. For instance, many electronic locks fail secure but accept a physical key during power failures. Similarly, fire escapes provide a redundant exit path for emergencies. While redundancy increases reliability, it also increases the attack surface. An adversary need only find a vulnerability in one path to be successful.

Reliability and Security Tradeoff: Incident Management

The presence of an adversary can also affect methods of collaboration and the information that's available to responders during an incident. Reliability incidents benefit from having responders with multiple perspectives who can help find and mitigate the root cause quickly. By contrast, you'll often want to handle security incidents with the smallest number of people who can fix the problem effectively, so the adversary isn't tipped off to the recovery effort. In the security case, you'll share information on a *need-to-know* basis. Similarly, voluminous system logs may inform the response to an incident and reduce your time to recovery, but—depending on what is logged—those logs may be a valuable target for an attacker.

Confidentiality, Integrity, Availability

Both security and reliability are concerned with the confidentiality, integrity, and availability of systems, but they view these properties through different lenses. The key difference between the two viewpoints is the presence or lack of a malicious adversary. A reliable system must not breach confidentiality accidentally, like a buggy chat system that misdelivers, garbles, or loses messages might. Additionally, a secure system must prevent an active adversary from accessing, tampering with, or destroying confidential data. Let's take a look at a few examples that demonstrate how a reliability problem can lead to a security issue.

Confidentiality, integrity, and availability have traditionally been considered fundamental attributes of secure systems and are referred to as the *CIA triad*. While many other models extend the set of security attributes beyond these three, the CIA triad has remained popular over time. Despite the acronym, this concept is not related to the Central Intelligence Agency in any way.

Confidentiality

In the aviation industry, having a push-to-talk microphone stuck in the transmit position is a notable confidentiality problem. In several well-documented cases, a stuck mike has broadcast private conversations between pilots in the cockpit, which represents a breach of confidentiality. In this case, no malicious adversary is involved: a hardware reliability flaw causes the device to transmit when the pilot does not intend it to.

Integrity

Similarly, data integrity compromise need not involve an active adversary. In 2015, Google Site Reliability Engineers (SREs) noticed that the end-to-end cryptographic integrity checks on a few blocks of data were failing. Because some of the machines that processed the data later demonstrated evidence of uncorrectable memory errors, the SREs decided to write software that exhaustively computed the integrity check for every version of the data with a single-bit flip (a 0 changed to a 1, or vice versa). That way, they could see if one of the results matched the value of the original integrity check. All errors indeed turned out to be single-bit flips, and the SREs recovered all the data. Interestingly, this was an instance of a security technique coming to the rescue during a reliability incident. (Google's storage systems also use noncryptographic end-to-end integrity checks, but other issues prevented SREs from detecting the bit flips.)

Availability

Finally, of course, availability is both a reliability and a security concern. An adversary might exploit a system's weak spot to bring the system to a halt or impair its operation for authorized users. Or they might control a large number of devices spread around the world to perform a classic distributed denial-of-service (DDoS) attack, instructing the many devices to flood a victim with traffic.

Denial-of-service (DoS) attacks are an interesting case because they straddle the areas of reliability and security. From a victim's point of view, a malicious attack may be indistinguishable from a design flaw or a legitimate spike in traffic. For example, a 2018 software update caused some Google Home and Chromecast devices to generate large synchronized spikes in network traffic as the devices adjusted their clocks,

resulting in unexpected load on Google's central time service. Similarly, a major breaking news story or other event that prompts millions of people to issue near-identical queries can look very much like a traditional application-level DDoS attack. As shown in Figure 1-1, when a magnitude 4.5 earthquake hit the San Francisco Bay Area in the middle of the night in October 2019, Google infrastructure serving the area was hit with a flood of queries.

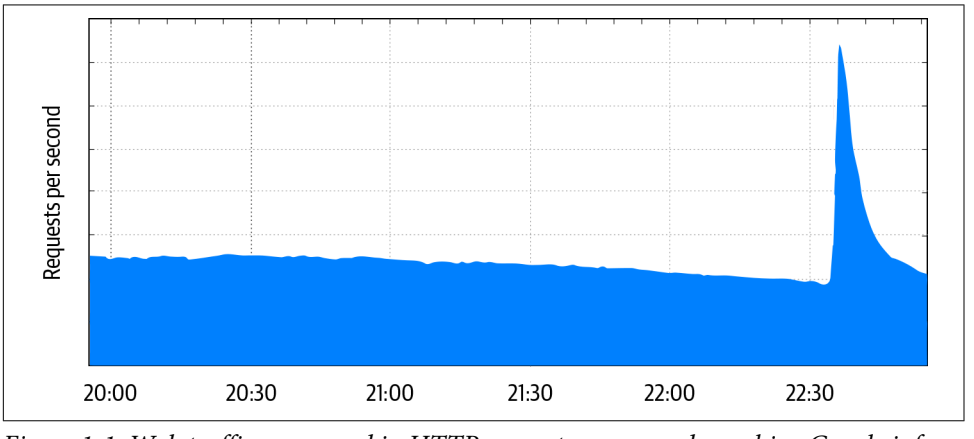

Figure 1-1. Web traffic, measured in HTTP requests per second, reaching Google infrastructure serving users in the San Francisco Bay Area when a magnitude 4.5 earthquake hit the region on October 14, 2019

Reliability and Security: Commonalities

Reliability and security—unlike many other system characteristics—are emergent properties of a system's design. Both are also difficult to bolt on after the fact, so you should ideally take both into account from the earliest design stages. They also require ongoing attention and testing throughout the entire system lifecycle, because it is easy for system changes to inadvertently affect them. In a complex system, reliability and security properties are often determined by the interaction of many components, and an innocuous-looking update to one component may end up affecting the reliability or security of the entire system in a way that may not be evident until it causes an incident. Let's examine these and other commonalities in more detail.

Invisibility

Reliability and security are mostly invisible when everything is going well. But one of the goals of reliability and security teams is to earn and keep the trust of customers and partners. Good communication—not only in times of trouble, but also when things are going well—is a solid foundation for this trust. It is important that the information be—to the greatest degree possible—honest and concrete, and free of platitudes and jargon.

Unfortunately, the inherent invisibility of reliability and security in the absence of emergencies means that they're often seen as costs that you can reduce or defer without immediate consequences. However, the costs of reliability and security failures can be severe. According to media reports, data breaches may have led to a $350 million reduction in the price Verizon paid to acquire Yahoo!'s internet business in 2017. In the same year, a power failure caused key computer systems to shut down at Delta Airlines and resulted in almost 700 flight cancellations and thousands of delays, reducing Delta's flight throughput for the day by approximately 60%.

Assessment

Because it's not practical to achieve perfect reliability or security, you can use risk-based approaches to estimate the costs of negative events, as well as the up-front and opportunity costs of preventing these events. However, you should measure the probability of negative events differently for reliability and security. You can reason about the reliability of a composition of systems and plan engineering work according to desired error budgets,[1] at least in part because you can assume independence of failures across the individual components. The security of such a composition is more difficult to assess. Analyzing a system's design and implementation can afford some level of assurance. Adversarial testing—simulated attacks typically performed from the perspective of a defined adversary—can also be used to evaluate a system's resistance to particular kinds of attacks, the effectiveness of attack detection mechanisms, and the potential consequences of attacks.

Simplicity

Keeping system design as simple as possible is one of the best ways to improve your ability to assess both the reliability and the security of a system. A simpler design reduces the attack surface, decreases the potential for unanticipated system interactions, and makes it easier for humans to comprehend and reason about the system. Understandability is especially valuable during emergencies, when it can help responders mitigate symptoms quickly and reduce mean time to repair (MTTR). Chapter 6 elaborates on this topic and discusses strategies such as minimizing attack surfaces and isolating responsibility for security invariants into small, simple subsystems that can be reasoned about independently.

1 For more information on error budgets, see Chapter 3 in the SRE book.

Evolution

No matter how simple and elegant the initial design, systems rarely remain unchanged over time. New feature requirements, changes in scale, and evolution of the underlying infrastructure all tend to introduce complexity. On the security side, the need to keep up with evolving attacks and new adversaries can also increase system complexity. Additionally, pressure to meet market demands can lead system developers and maintainers to cut corners and accumulate technical debt. Chapter 7 addresses some of these challenges.

Complexity often accumulates inadvertently, but this can lead to tipping-point situations where a small and apparently innocuous change has major consequences for a system's reliability or security. A bug that was introduced in 2006 and discovered almost two years later in the Debian GNU/Linux version of the OpenSSL library provides one notorious example of a major failure caused by a small change. An open source developer noticed that Valgrind, a standard tool for debugging memory problems, was reporting warnings about memory used prior to initialization. To eliminate the warnings, the developer removed two lines of code. Unfortunately, this caused OpenSSL's pseudo-random number generator to only be seeded with a process ID, which on Debian at the time defaulted to a number between 1 and 32,768. Brute force could then easily break cryptographic keys.

Google has not been immune to failures triggered by seemingly innocuous changes. For example, in October 2018, YouTube was down globally for more than an hour because of a small change in a generic logging library. A change intended to improve the granularity of event logging looked innocent to both its author and the designated code reviewer, and it passed all tests. However, the developers didn't fully realize the impact of the change at YouTube scale: under production load, the change quickly caused YouTube servers to run out of memory and crash. As the failures shifted user traffic toward other, still healthy servers, cascading failures brought the entire service to a halt.

Resilience

Of course, a memory utilization problem should not have caused a global service outage. Systems should be designed to be resilient under adverse or unexpected circumstances. From the reliability perspective, such circumstances are often caused by unexpectedly high load or component failures. Load is a function of the volume and the average cost of requests to the system, so you can achieve resilience by shedding some of the incoming load (processing less) or reducing the processing cost for each request (processing more cheaply). To address component failures, system design should incorporate redundancy and distinct failure domains so that you can limit the impact of failures by rerouting requests. Chapter 8 discusses these topics further, and Chapter 10 goes into depth on DoS mitigations in particular.

However resilient a system's individual components might be, once it becomes sufficiently complex, you cannot easily demonstrate that the entire system is immune to compromise. You can address this problem in part using defense in depth and distinct failure domains. *Defense in depth* is the application of multiple, sometimes redundant, defense mechanisms. *Distinct failure domains* limit the "blast radius" of a failure and therefore also increase reliability. A good system design limits an adversary's ability to exploit a compromised host or stolen credentials in order to move laterally or to escalate privilege and affect other parts of the system.

You can implement distinct failure domains by compartmentalizing permissions or restricting the scope of credentials. For example, Google's internal infrastructure supports credentials that are explicitly scoped to a geographic region. These types of features can limit the ability of an attacker who compromises a server in one region to move laterally to servers in other regions.

Employing independent encryption layers for sensitive data is another common mechanism for defense in depth. For example, even though disks provide device-level encryption, it's often a good idea to also encrypt the data at the application layer. This way, even a flawed implementation of an encryption algorithm in a drive controller won't be sufficient to compromise the confidentiality of protected data if an attacker gains physical access to a storage device.

While the examples cited so far hinge on external attackers, you must also consider potential threats from malicious insiders. Although an insider may know more about potential abuse vectors than an external attacker who steals an employee's credentials for the first time, the two cases often don't differ much in practice. The *principle of least privilege* can mitigate insider threats. It dictates that a user should have the minimal set of privileges required to perform their job at a given time. For example, mechanisms like Unix's `sudo` support fine-grained policies that specify which users can run which commands as which role.

At Google, we also use multi-party authorization to ensure that sensitive operations are reviewed and approved by specific sets of employees. This multi-party mechanism both protects against malicious insiders and reduces the risk of innocent human error, a common cause of reliability failures. Least privilege and multi-party authorization are not new ideas—they have been employed in many noncomputing scenarios, from nuclear missile silos to bank vaults. Chapter 5 discusses these concepts in depth.

From Design to Production

Security and reliability considerations should be kept in mind when translating even a solid design into a fully deployed production system. Starting with the development of the code, opportunities exist to spot potential security and reliability issues

through code reviews, and even to prevent entire classes of problems by using common frameworks and libraries. Chapter 12 discusses some of these techniques.

Before deploying a system, you can use testing to ensure that it functions correctly both in normal scenarios and in the edge cases that typically impact reliability and security. Whether you use load testing to understand the behavior of a system under a flood of queries, fuzzing to explore the behavior on potentially unexpected inputs, or specialized tests to ensure that cryptographic libraries aren't leaking information, testing plays a critical role in gaining assurance that the system you've actually built matches your design intentions. Chapter 13 covers these approaches in depth.

Finally, some approaches to actually deploying code (see Chapter 14) can limit security and reliability risk. For example, canaries and slow rollouts can prevent you from breaking the system for all users simultaneously. Similarly, a deployment system that accepts only code that's been properly reviewed can help to mitigate the risk of an insider pushing a malicious binary to production.

Investigating Systems and Logging

So far we have focused on design principles and implementation approaches to prevent both reliability and security failures. Unfortunately, it is usually impractical or too expensive to attain perfect reliability or security. You must assume that preventive mechanisms will fail, and craft a plan to detect and recover from failures.

As we discuss in Chapter 15, good logging is the foundation of detection and failure preparedness. In general, the more complete and detailed your logs, the better—but this guideline has some caveats. At sufficient scale, log volume poses a significant cost, and analyzing logs effectively can become difficult. The YouTube example from earlier in this chapter illustrates that logging can also introduce reliability problems. Security logs pose an additional challenge: logs typically should not contain sensitive information, such as authentication credentials or personally identifiable information (PII), lest the logs themselves become attractive targets for adversaries.

Crisis Response

During an emergency, teams must work together quickly and smoothly because problems can have immediate consequences. In the worst case, an incident can destroy a business in minutes. For example, in 2014 an attacker put the code-hosting service Code Spaces out of business in a matter of hours by taking over the service's administrative tools and deleting all of its data, including all backups. Well-rehearsed collaboration and good incident management are critical for timely responses in these situations.

Organizing crisis response is challenging, so it's best to have a plan in place before an emergency occurs. By the time you discover an incident, the clock may have been

ticking for some time. In any case, responders are operating under stress and time pressure, and (at least initially) with limited situational awareness. If an organization is large and the incident requires 24/7 response capabilities or collaboration across time zones, the need to maintain state across teams and to hand off incident management at the boundaries of work shifts further complicates operations. Security incidents also typically entail tension between the impulse to involve all essential parties versus the need—often driven by legal or regulatory requirements—to restrict information sharing on a need-to-know basis. Moreover, the initial security incident may be just the tip of the iceberg. The investigation might grow beyond company boundaries or involve law enforcement agencies.

During a crisis, it is essential to have a clear chain of command and a solid set of checklists, playbooks, and protocols. As discussed in Chapters 16 and 17, Google has codified crisis response into a program called Incident Management at Google (IMAG), which establishes a standard, consistent way to handle all types of incidents, from system outages to natural disasters, and organize an effective response. IMAG was modeled on the US government's Incident Command System (ICS), a standardized approach to the command, control, and coordination of emergency response among responders from multiple government agencies.

When not faced with the pressure of an ongoing incident, responders typically negotiate long intervals with little activity. During these times, teams need to keep individuals' skills and motivation sharp and improve processes and infrastructure in preparation for the next emergency. Google's Disaster Recovery Testing program (DiRT) regularly simulates various internal system failures and forces teams to cope with these types of scenarios. Frequent offensive security exercises test our defenses and help identify new vulnerabilities. Google employs IMAG even for small incidents, which further prompts us to regularly exercise emergency tools and processes.

Recovery

Recovering from a security failure often requires patching systems to fix a vulnerability. Intuitively, you want that process to happen as quickly as possible, using mechanisms that are exercised regularly and are therefore decently reliable. However, the capability to push changes quickly is a double-edged sword: while this capability can help close vulnerabilities quickly, it can also introduce bugs or performance issues that cause a lot of damage. The pressure to push patches quickly is greater if the vulnerability is widely known or severe. The choice of whether to push fixes slowly—and therefore to have more assurance that there are no inadvertent side effects, but risk that the vulnerability will be exploited—or to do so quickly ultimately comes down to a risk assessment and a business decision. For example, it may be acceptable to lose some performance or increase resource usage to fix a severe vulnerability.

Choices like this underscore the need for reliable recovery mechanisms that allow us to roll out necessary changes and updates quickly without compromising reliability, and that also catch potential problems before they cause a widespread outage. For example, a robust fleet recovery system needs to have a reliable representation of every machine's current and desired state, and also needs to provide backstops to ensure that state is never rolled back to an obsolete or unsafe version. Chapter 9 covers this and many other approaches, and Chapter 18 discusses how to actually recover systems once an event has occurred.

Conclusion

Security and reliability have a lot in common—both are inherent properties of all information systems that are tempting to initially sacrifice in the name of velocity, but costly to fix after the fact. This book aims to help you address inevitable challenges related to security and reliability early on, as your systems evolve and grow. Alongside engineering efforts, each organization has to understand the roles and responsibilities (see Chapter 20) that contribute to building a culture of security and reliability (Chapter 21) in order to persist sustainable practices. By sharing our experiences and lessons learned, we hope to enable you to avoid paying a bigger price further down the road by adopting some of the principles described here sufficiently early in the system lifecycle.

We wrote this book with a broad audience in mind, with the goal that you will find it relevant regardless of the stage or scope of your project. While reading it, keep the risk profile of your project in mind—operating a stock exchange or a communication platform for dissidents has a drastically different risk profile than running a website for an animal sanctuary. The next chapter discusses the classes of adversaries and their possible motivations in detail.