CHAPTER 7
Design for a Changing Landscape

By Maya Kaczorowski, John Lunney, and Deniz Pecel
with Jen Barnason, Peter Duff, and Emily Stark

Your ability to adapt while fulfilling the service level objectives promised to your users depends on the robustness and flexibility of your service's reliability capabilities. Tools and approaches like frequent builds, releases with automated testing, and containers and microservices will enable you to adapt to short-, medium-, and long-term changes, as well as unexpected complications that arise as you run a service. This chapter also presents multiple examples of how Google has changed and adapted its systems over the years, and the lessons we learned along the way.

While most design decisions—particularly those relating to architecture—are easiest and cheapest to implement in the system design phase, many of the best practices in this chapter can be implemented during later stages of the system lifecycle.

"Change is the only constant" is a maxim[1] that certainly holds true for software: as the number (and variety) of devices we use increases every year, so too does the number of library and application vulnerabilities. Any device or application is potentially susceptible to remote exploit, data leakage, botnet takeover, or other headline-grabbing scenarios.

At the same time, the security and privacy expectations of users and regulators continue to rise, requiring stricter controls like enterprise-specific access restrictions and authentication systems.

1 Widely attributed to Heraclitus of Ephesus.

To respond to this shifting landscape of vulnerabilities, expectations, and risks, you need to be able to change your infrastructure frequently and quickly, while also maintaining a highly reliable service—not an easy feat. Achieving this balance often boils down to deciding when, and how quickly, to roll out a change.

Types of Security Changes

There are many kinds of changes you might make to improve your security posture or the resilience of your security infrastructure—for example:

- Changes in response to security incidents (see Chapter 18)
- Changes in response to newly discovered vulnerabilities
- Product or feature changes
- Internally motivated changes to improve your security posture
- Externally motivated changes, such as new regulatory requirements

Some types of security-motivated changes require additional considerations. If you're rolling out a feature optionally as a first step toward making it mandatory, you'll need to collect sufficient feedback from early adopters and thoroughly test your initial instrumentation.

If you're considering a change to a dependency—for example, a vendor or third-party code dependency—you'll need to make sure the new solution meets your security requirements.

Designing Your Change

Security changes are subject to the same basic reliability requirements and release engineering principles as any other software changes; for more information, see Chapter 4 in this book and Chapter 8 in the SRE book. The timeline for rolling out security changes may differ (see "Different Changes: Different Speeds, Different Timelines" on page 127), but the overall process should follow the same best practices.

All changes should have the following characteristics:

Incremental
: Make changes that are as small and standalone as possible. Avoid the temptation to tie a change to unrelated improvements, such as refactoring code.

Documented
: Describe both the "how" and the "why" of your change so others can understand the change and the relative urgency of the rollout. Your documentation might include any or all of the following:

- Requirements
- Systems and teams in scope
- Lessons learned from a proof of concept
- Rationale for decisions (in case plans need to be reevaluated)
- Points of contact for all teams involved

Tested
: Test your security change with unit tests and—where possible—integration tests (for more information on testing, see Chapter 13). Complete a peer review to gain a measure of confidence that the change will work in production.

Isolated
: Use feature flags to isolate changes from one another and avoid release incompatibility; for more information, see Chapter 16 in the SRE workbook. The underlying binary should exhibit no change in behavior when the feature is turned off.

Qualified
: Roll out your change with your normal binary release process, proceeding through stages of qualification before receiving production or user traffic.

Staged
: Roll out your change gradually, with instrumentation for canarying. You should be able to see differences in behavior before and after your change.

These practices suggest taking a "slow and steady" approach to rollout. In our experience, the conscious tradeoff between speed and safety is worthwhile. You don't want to risk creating a much larger problem like widespread downtime or data loss by rolling out a broken change.

Architecture Decisions to Make Changes Easier

How can you architect your infrastructure and processes to be responsive to the inevitable changes you'll face? Here we discuss some strategies that enable you to flex your system and roll out changes with minimal friction, which also lead to building a culture of security and reliability (discussed in Chapter 21).

Keep Dependencies Up to Date and Rebuild Frequently

Making sure your code points to the latest versions of code dependencies helps make your system less susceptible to new vulnerabilities. Keeping references to dependencies up to date is particularly important for open source projects that change often, like OpenSSL or the Linux kernel. Many large open source projects have

well-established security vulnerability response and remediation plans that clarify when a new release contains a critical security patch, and will backport the fix to supported versions. If your dependencies are up to date, it's likely you can apply a critical patch directly instead of needing to merge with a backlog of changes or apply multiple patches.

New releases and their security patches won't make it into your environment until you rebuild. Frequently rebuilding and redeploying your environment means that you'll be ready to roll out a new version when you need to—and that an emergency rollout can pick up the latest changes.

Release Frequently Using Automated Testing

Basic SRE principles recommend cutting and rolling out releases regularly to facilitate emergency changes. By splitting one large release into many smaller ones, you ensure that each release contains fewer changes, which are therefore less likely to require rollback. For a deeper exploration of this topic, see the "virtuous cycle" depicted in Figure 16-1 in the SRE workbook.

When each release contains fewer code changes, it's easier to understand what changed and pinpoint potential issues. When you need to roll out a security change, you can be more confident about the expected outcome.

To take full advantage of frequent releases, automate their testing and validation. This allows good releases to be pushed automatically while preventing deficient releases from reaching production. Automated testing also gives you additional confidence when you need to push out fixes that protect against critical vulnerabilities.

Similarly, by using containers[2] and microservices,[3] you can reduce the surface area you need to patch, establish regular release processes, and simplify your understanding of system vulnerabilities.

Use Containers

Containers decouple the binaries and libraries your application needs from the underlying host OS. Because each application is packaged with its own dependencies and libraries, the host OS does not need to include them, so it can be smaller. As a result, applications are more portable and you can secure them independently. For

2 For more information on containers, see the blog post "Exploring Container Security" by Dan Lorenc and Maya Kaczorowski. See also Burns, Brendan et al. 2016. "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems Over a Decade." *ACM Queue* 14(1). *https://oreil.ly/tDKBJ*.

3 For more information on microservices, see "Case Study 4: Running Hundreds of Microservices on a Shared Platform" in Chapter 7 of the SRE book.

example, you can patch a kernel vulnerability in the host operating system without having to change your application container.

Containers are meant to be immutable, meaning they don't change after they're deployed—instead of SSHing into a machine, you rebuild and redeploy the whole image. Because containers are short-lived, they're rebuilt and redeployed quite often.

Rather than patching live containers, you patch the images in your container registry. This means that you can roll out a fully patched container image as one unit, making the patch rollout process the same as your (very frequent) code rollout process—complete with monitoring, canarying, and testing. As a result, you can patch more often.

As these changes roll out to each task, the system seamlessly moves serving traffic to another instance; see "Case Study 4: Running Hundreds of Microservices on a Shared Platform" in Chapter 7 of the SRE book. You can achieve similar results and avoid downtime while patching with blue/green deployments; see Chapter 16 in the SRE workbook.

You can also use containers to detect and patch newly discovered vulnerabilities. Since containers are immutable, they provide content addressability. In other words, you actually know what's running in your environment—for example, which images you've deployed. If you previously deployed a fully patched image that happens to be susceptible to a new vulnerability, you can use your registry to identify the susceptible versions and apply patches, rather than scanning your production clusters directly.

To reduce the need for this kind of ad hoc patching, you should monitor the age of containers running in production and redeploy regularly enough to ensure that old containers aren't running. Similarly, to avoid redeploying older, unpatched images, you should enforce that only recently built containers can be deployed in production.

Use Microservices

An ideal system architecture is easily scalable, provides visibility into system performance, and allows you to manage each potential bottleneck between services in your infrastructure. Using a microservices architecture, you can split workloads into smaller, more manageable units to facilitate maintenance and discovery. As a result, you can independently scale, load balance, and perform rollouts in each microservice, which means you have more flexibility to make infrastructure changes. Since each service handles requests separately, you can use several defenses independently and sequentially, providing defense in depth (see "Defense in Depth" on page 145).

Microservices also naturally facilitate limited or zero trust networking, meaning that your system doesn't inherently trust a service just because it's located in the same network (see Chapter 6). Rather than using a perimeter-based security model with untrusted external versus trusted internal traffic, microservices use a more

heterogeneous notion of trust inside the perimeter: internal traffic may have different levels of trust. Current trends are moving toward an increasingly segmented network. As dependence on a single network perimeter, like a firewall, is removed, the network can be further segmented by services. At the extreme end, a network can implement microservice-level segmentation with no inherent trust between services.

A secondary consequence of using microservices is the convergence of security tools, so that some processes, tools, and dependencies can be reused across multiple teams. As your architecture scales, it might make sense to consolidate your efforts to address shared security requirements—for example, by using common cryptographic libraries or common monitoring and alerting infrastructure. That way, you can split critical security services into separate microservices that are updated and managed by a small number of responsible parties. It's important to note that achieving the security advantages of microservices architectures requires restraint to ensure that the services are as simple as possible while still maintaining the desired security properties.

Using a microservices architecture and development process allows teams to address security issues early in the development and deployment lifecycle—when it's less costly to make changes—in a standardized way. As a result, developers can achieve secure outcomes while spending less time on security.

Example: Google's frontend design

Google's frontend design uses microservices to provide resilience and defense in depth.[4] Separating the frontend and backend into different layers has many advantages: Google Front End (GFE) serves as a frontend layer to most Google services, implemented as a microservice, so these services aren't directly exposed to the internet. GFE also terminates traffic for incoming HTTP(S), TCP, and TLS proxies; provides DDoS attack countermeasures; and routes and load balances traffic to Google Cloud services.[5]

GFE allows for independent partitioning of frontend and backend services, which has benefits in terms of scalability, reliability, agility, and security:

- Global load balancing helps move traffic between GFE and backends. For example, we can redirect traffic during a datacenter outage, reducing mitigation time.
- Backend and frontend layers can have several layers within themselves. Because each layer is a microservice, we can load balance each layer. As a result, it's relatively easy to add capacity, make general changes, or apply rapid changes to each microservice.

4 See Chapter 2 in the SRE book.

5 See the "Encryption in Transit in Google Cloud" whitepaper for more information.

- If a service becomes overloaded, GFE can serve as a mitigation point by dropping or absorbing connections before the load reaches the backends. This means that not every layer in a microservices architecture needs its own load protection.
- Adoption of new protocols and security requirements is relatively straightforward. GFE can handle IPv6 connections even if some backends aren't yet ready for them. GFE also simplifies certificate management by serving as the termination point for various common services, like SSL.[6] For example, when a vulnerability was discovered in the implementation of SSL renegotiations, GFE's control for limiting these renegotiations protected all the services behind it. Rapid Application Layer Transport Security encryption adoption also illustrates how a microservices architecture facilitates change adoption: Google's security team integrated the ALTS library into its RPC library to handle service credentials, which enabled wide adoption without a significant burden on individual development teams.

In today's cloud world, you can achieve benefits similar to those described here by using a microservices architecture, building layers of security controls, and managing cross-service communications with a service mesh. For example, you might separate request processing from the configuration for managing request processing. The industry refers to this type of deliberate split as *separation of the data plane* (the requests) *and the control plane* (the configuration). In this model, the data plane provides the actual data processing in the system, typically handling load balancing, security, and observability. The control plane provides policy and configuration to the data plane services, thereby providing a manageable and scalable control surface.

Different Changes: Different Speeds, Different Timelines

Not all changes occur on the same timelines or at the same speed. Several factors influence how quickly you might want to make a change:

Severity
: Vulnerabilities are discovered every day, but not all of them are critical, actively being exploited, or applicable to your particular infrastructure. When you *do* hit that trifecta, you likely want to release a patch as soon as possible. Accelerated timelines are disruptive and more likely to break systems. Sometimes speed is necessary, but it's generally safer for a change to happen slowly so you can ensure sufficient product security and reliability. (Ideally, you can apply a critical security patch independently—that way, you can apply the patch quickly without unnecessarily accelerating any other in-flight rollouts.)

6 Ibid.

Dependent systems and teams
: Some system changes may be dependent on other teams that need to implement new policies or enable a particular feature prior to rollout. Your change may also depend on an external party—for example, if you need to receive a patch from a vendor, or if clients need to be patched before your server.

Sensitivity
: The sensitivity of your change may affect when you can deploy it to production. A nonessential change that improves an organization's overall security posture isn't necessarily as urgent as a critical patch. You can roll out that nonessential change more gradually—for example, team by team. Depending on other factors, making the change may not be worth the risk—for example, you may not want to roll out a nonurgent change during critical production windows like a holiday shopping event, where changes are otherwise tightly controlled.

Deadline
: Some changes have a finite deadline. For example, a regulatory change might have a specified compliance date, or you may need to apply a patch before a news embargo (see the following sidebar) disclosing a vulnerability drops.

Embargoed Vulnerabilities

Known vulnerabilities that haven't been disclosed via public announcement can be tricky to handle. Information that's *under embargo* can't be released before a certain date, which might be the date that a patch is going to be made available or rolled out, or the date a researcher plans to disclose the issue.

If you are privy to information about a vulnerability under embargo, and rolling out a patch would break the embargo, you must wait for a public announcement before you can patch along with the rest of the industry. If you're involved in incident response prior to the announcement of a vulnerability, work with other parties to agree on an announcement date that suits the rollout processes of most organizations—for example, a Monday.

There is no hard-and-fast rule for determining the speed of a particular change—a change that requires a quick configuration change and rollout in one organization may take months in another organization. While a single team may be able to make a given change according to a specific timeline, there may be a long tail for your organization to fully adopt the change.

In the following sections, we discuss three different time horizons for change and include examples to show what each has looked like at Google:

- A short-term change in reaction to a new security vulnerability
- A medium-term change, where new product adoption could happen gradually
- A long-term change for regulatory reasons, where Google had to build new systems in order to implement the change

Short-Term Change: Zero-Day Vulnerability

Newly discovered vulnerabilities often require short-term action. A *zero-day vulnerability* is one that is known by at least some attackers, but that hasn't been disclosed publicly or discovered by the targeted infrastructure provider. Typically, a patch either isn't available yet or hasn't been widely applied.

There are a variety of ways to find out about new vulnerabilities that might affect your environment, including regular code reviews, internal code scanning (see "Sanitize Your Code" on page 267), fuzzing (see "Fuzz Testing" on page 280), external scans like penetration tests and infrastructure scans, and bug bounty programs.

In the context of short-term changes, we'll focus on vulnerabilities where Google learned about the vulnerability on day zero. Although Google is often involved in embargoed vulnerability responses—for example, when developing patches—a short-term change for a zero-day vulnerability is common behavior for most organizations in the industry.

Although zero-day vulnerabilities get a lot of attention (both externally and within the organization), they're not necessarily the vulnerabilities that are most exploited by attackers. Before you tackle a same-day zero-day vulnerability response, make sure you're patched for the "top hits" to cover critical vulnerabilities from recent years.

When you discover a new vulnerability, triage it to determine its severity and impact. For example, a vulnerability that allows remote code execution may be considered critical. But the impact to your organization might be very difficult to determine: Which systems use this particular binary? Is the affected version deployed in production? Where possible, you'll also want to establish ongoing monitoring and alerting to determine if the vulnerability is being actively exploited.

To take action, you need to obtain a *patch*—a new version of the affected package or library with a fix applied. Begin by verifying that the patch actually addresses the vulnerability. It can be useful to do this using a working exploit. However, be aware that even if you can't trigger the vulnerability with the exploit, your system might still be vulnerable (recall that the absence of evidence is not evidence of absence). For

example, the patch you've applied might address only one possible exploit of a larger class of vulnerabilities.

Once you've verified your patch, roll it out—ideally in a test environment. Even on an accelerated timeline, a patch should be rolled out gradually like any other production change—using the same testing, canarying, and other tools—on the order of hours or days.[7] A gradual rollout allows you to catch potential issues early, as the patch may have an unexpected effect on your applications. For example, an application using an API you were unaware of may impact performance characteristics or cause other errors.

Sometimes you can't directly fix the vulnerability. In this case, the best course of action is to mitigate the risk by limiting or otherwise restricting access to the vulnerable components. This mitigation may be temporary until a patch is available, or permanent if you can't apply the patch to your system—for example, because of performance requirements. If suitable mitigations are already in place to secure your environment, you may not even need to take any further action.

For more details on incident response, see Chapter 17.

Example: Shellshock

On the morning of September 24, 2014, Google Security learned about a publicly disclosed, remotely exploitable vulnerability in bash that trivially allowed code execution on affected systems. The vulnerability disclosure was quickly followed by exploits in the wild, starting the same day.

The original report had some muddled technical details and didn't clearly address the status of the embargo on discussing the issue. This report, in addition to the rapid discovery of several similar vulnerabilities, caused confusion about the nature and exploitability of the attack. Google's Incident Response team initiated its Black Swan protocol to address an exceptional vulnerability and coordinated a large-scale response to do the following:[8]

- Identify the systems in scope and the relative risk level for each
- Communicate internally to all teams that were potentially affected
- Patch all vulnerable systems as quickly as possible
- Communicate our actions, including remediation plans, externally to partners and customers

7 See the discussion of gradual and staged rollouts in Chapter 27 of the SRE book.

8 See also the YouTube video of the panel discussion about this event.

We were not aware of the issue before public disclosure, so we treated it as a zero-day vulnerability that necessitated emergency mitigation. In this case, a patched version of bash was already available.

The team assessed the risk to different systems and acted accordingly:

- We deemed a huge number of Google production servers to be low risk. These servers were easy to patch with an automated rollout. Once servers passed sufficient validation and testing, we patched them much faster than usual, rather than in long phases.
- We deemed a large number of Googler workstations to be higher risk. Fortunately, these workstations were easy to patch quickly.
- A small number of nonstandard servers and inherited infrastructure were deemed high risk and needed manual intervention. We sent notifications to each team detailing the follow-up actions they were required to take, which allowed us to scale the effort to multiple teams and easily track progress.

In parallel, the team developed software to detect vulnerable systems within Google's network perimeter. We used this software to complete the remaining patch work needed, and added this functionality to Google's standard security monitoring.

What we did (and didn't) do well during this response effort offers a number of lessons for other teams and organizations:

- *Standardize software distribution to the greatest extent possible*, so that patching is the easy and simple choice for remediation. This also requires service owners to understand and accept the risk of choosing a nonstandard, nonsupported distribution. A service owner should be responsible for maintaining and patching an alternative option.
- *Use public distribution standards*—ideally, the patch you need to roll out will already be in the right format. That way, your team can start validating and testing the patch quickly, rather than needing to rework the patch to address your specific environment.
- *Ensure that you can accelerate your mechanism to push changes for emergency changes* like zero-day vulnerabilities. This mechanism should allow for faster than usual validation before full rollout to the affected systems. We don't necessarily recommend that you skip validating that your environment still functions—you must balance this step against the need to mitigate the exploit.
- *Make sure that you have monitoring to track the progress of your rollout, that you identify unpatched systems, and that you identify where you're still vulnerable*. If you already have tooling to identify whether a vulnerability is currently being

exploited in your environment, it may help you decide to slow down or speed up based on your current risk.

- *Prepare external communications as early in your response efforts as possible*. You don't want to get bogged down in internal PR approvals when the media is calling for a response.
- *Draft a reusable incident or vulnerability response plan* (see Chapter 17) ahead of time, including language for external communications. If you're not sure what you need, start with the postmortem of a previous event.
- *Know which systems are nonstandard or need special attention*. By keeping track of outliers, you'll know which systems might need proactive notifications and patching assistance. (If you standardize software distribution per our advice in bullet one, outliers should be limited.)

Medium-Term Change: Improvement to Security Posture

Security teams often implement changes to improve an environment's overall security posture and reduce risk. These proactive changes are driven by internal and external requirements and deadlines, and rarely need to be rolled out suddenly.

When planning for changes to your security posture, you need to figure out which teams and systems are affected and determine the best place to start. Following the SRE principles outlined in "Designing Your Change" on page 122, put together an action plan for gradual rollout. Each phase should include success criteria that must be met before moving to the next phase.

Systems or teams affected by security changes can't necessarily be represented as a percentage of a rollout. You can instead phase a rollout according to who is affected and what changes should be made.

In terms of who is affected, roll out your change group by group, where a group might be a development team, system, or set of end users. For example, you might begin by rolling out a change for device policies to users who are frequently on the road, like your sales team. Doing so allows you to quickly test the most common cases and obtain real-world feedback. There are two competing philosophies when it comes to rollout populations:

- Start with the easiest use case, where you'll get the most traction and prove value.
- Start with the hardest use case, where you'll find the most bugs and edge cases.

When you're still seeking buy-in from the organization, it makes sense to start with the easiest use case. If you have leadership support and investment up front, it's more valuable to find implementation bugs and pain points early on. In addition to organizational concerns, you should consider which strategy will lead to the greatest risk

reduction, both in the short and the long term. In all cases, a successful proof of concept helps determine how to best move forward. The team making the change should also have to live through it, "eating their own dogfood," so that they understand the user experience.

You might also be able to roll out the change itself incrementally. For example, you may be able to implement progressively more stringent requirements, or the change could initially be opt-in, rather than mandatory. Where possible, you should also consider rolling out a change as a dry run in an alerting or auditing mode before switching to an enforcement mode—that way, users can experience how they'll be affected before the change is mandatory. This allows you to find users or systems that you've improperly identified as in scope, as well as users or systems for whom achieving compliance will be particularly difficult.

Example: Strong second-factor authentication using FIDO security keys

Phishing is a significant security concern at Google. Although we have widely implemented second-factor authentication using one-time passwords (OTPs), OTPs are still susceptible to interception as part of a phishing attack. We assume that even the most sophisticated users are well intentioned and well prepared to deal with phishing, but still susceptible to account takeovers due to confusing user interfaces or user error. To address this risk, starting in 2011, we investigated and tested several stronger two-factor authentication (2FA) methods.[9] We eventually chose universal two-factor (U2F) hardware security tokens because of their security and usability properties. Implementing security keys for Google's large and globally distributed employee population required building custom integrations and coordinating en masse enrollment.

Evaluating potential solutions and our final choice was part of the change process itself. Up front, we defined security, privacy, and usability requirements. We then validated potential solutions with users to understand what was changing, get real-world feedback, and measure the impact of the change on day-to-day workflows.

In addition to security and privacy requirements, the potential solution had to meet usability requirements to facilitate seamless adoption, also critical for building a culture of security and reliability (see Chapter 21). 2FA needed to be easy—fast and "brainless" enough to make using it incorrectly or insecurely difficult. This requirement was particularly critical for SREs—in case of outages, 2FA couldn't slow down response processes. Additionally, internal developers needed to be able to easily integrate the 2FA solution into their websites through simple APIs. In terms of ideal

9 See Lang, Juan et al. 2016. "Security Keys: Practical Cryptographic Second Factors for the Modern Web." *Proceedings of the 2016 International Conference on Financial Cryptography and Data Security*: 422–440. https://oreil.ly/S2ZMU.

usability requirements, we wanted an efficient solution that scaled for users across multiple accounts and didn't entail additional hardware, and that was physically effortless, easy to learn, and easy to recover if lost.

After evaluating several options, we co-designed FIDO security keys. Though these keys did not meet all of our ideal requirements, in initial pilots security keys decreased the total authentication time and had a negligible authentication failure rate.

Once we had a solution, we had to roll out security keys to all users and deprecate OTP support across Google. We began rolling out the security keys in 2013. To ensure wide adoption, enrollment was self-service:

- Initially, many users opted into security keys voluntarily because the keys were simpler to use than the existing OTP tools—they didn't have to type in a code from their phone or use a physical OTP device. Users were given "nano" security keys that could stay in the USB drives of their laptops.
- To obtain a security key, a user could go to any TechStop location in any office.[10] (Distributing the devices to global offices was complicated, requiring legal teams for export compliance and customs import requirements.)
- Users enrolled their security keys via a self-service registration website. TechStops provided assistance for the very first adopters and people who needed additional help. Users needed to use the existing OTP system the first time they authenticated, so keys were trusted on first use (TOFU).
- Users could enroll multiple security keys so they wouldn't need to worry about losing their key. This approach increased the overall cost, but was strongly aligned with our goal of not making the user carry additional hardware.

The team did encounter some issues, like out-of-date firmware, during rollout. When possible, we approached these issues in a self-service manner—for example, by allowing users to update security key firmware themselves.

Making security keys accessible to users was only half the problem, though. Systems using OTPs also needed to convert to using security keys. In 2013, many applications did not natively support this recently developed technology. The team first focused on supporting applications used daily by Googlers, like internal code review tools and dashboards. Where security keys were not supported (for example, in the case of some hardware device certificate management and third-party web applications), Google worked directly with the vendor to request and add support. We then had to deal with the long tail of applications. Since all OTPs were centrally generated, we

10 TechStops are Google's IT helpdesks, described in a blog post by Jesus Lugo and Lisa Mauck.

could figure out which application to target next by tracking the clients making OTP requests.

In 2015, the team focused on completing the rollout and deprecating the OTP service. We sent users reminders when they used an OTP instead of a security key, and eventually blocked access via OTP. Though we had dealt with most of the long tail of OTP application needs, there were still a few exceptions, such as mobile device setup. For these cases, we created a web-based OTP generator for exceptional circumstances. Users were required to verify their identity with their security key—a reasonable failure mode with a slightly higher time burden. We successfully completed the company-wide rollout of security keys in 2015.

This experience provided several generally applicable lessons, relevant for building a culture of security and reliability (see Chapter 21):

Make sure the solution you choose works for all users.
: It was critical that the 2FA solution was accessible so that visually impaired users weren't excluded.

Make the change easy to learn and as effortless as possible.
: This especially applies if the solution is more user-friendly than the initial situation! This is particularly important for an action or change you expect a user to perform frequently, where a little bit of friction can lead to a significant user burden.

Make the change self-service in order to minimize the burden on a central IT team.
: For a widespread change that affects all users in everyday activities, it is important that they can easily enroll, unenroll, and troubleshoot issues.

Give the user tangible proof that the solution works and is in their best interest.
: Clearly explain the impact of the change in terms of security and risk reduction, and provide an opportunity for them to offer feedback.

Make the feedback loop on policy noncompliance as fast as possible.
: This feedback loop can be an authentication failure, a system error, or an email reminder. Letting the user know that their action was not in line with the desired policy within minutes or hours allows them to take action to fix the issue.

Track progress and determine how to address the long tail.
: By examining user requests for OTPs by application, we could identify which applications to focus on next. Use dashboards to track progress and identify whether alternative solutions with similar security properties can work for the long tail of use cases.

Long-Term Change: External Demand

In some situations, you either have or need much more time to roll out a change—for example, an internally driven change that requires significant architectural or system changes, or a broader industry-wide regulatory change. These changes may be motivated or restricted by external deadlines or requirements, and might take several years to implement.

When taking on a massive, multiyear effort, you need to clearly define and measure progress against your goal. Documentation is particularly critical, both to ensure you take necessary design considerations into account (see "Designing Your Change" on page 122) and to maintain continuity. Individuals working on the change today might leave the company and need to hand off their work. Keeping documentation up to date with the latest plan and status is important for sustaining ongoing leadership support.

To measure ongoing progress, establish appropriate instrumentation and dashboarding. Ideally, a configuration check or test can measure a change automatically, removing the need to have a human in the loop. Just as you strive for significant test coverage for code in your infrastructure, you should aim for compliance check coverage for systems affected by the change. To scale this coverage effectively, this instrumentation should be self-serve, allowing teams to implement both the change and the instrumentation. Tracking these results transparently helps motivate users and simplifies communications and internal reporting. Rather than duplicating work, you should also use this single source of truth for executive communications.

Conducting any large-scale, long-term change in an organization while maintaining continued leadership support is difficult. To sustain momentum over time, the individuals making these changes need to stay motivated. Establishing finite goals, tracking progress, and demonstrating significant examples of impact can help teams finish the marathon. Implementation will always have a long tail, so figure out a strategy that makes the most sense for your situation. If a change is not required (by regulation, or for other reasons), achieving 80% or 90% adoption can have a measurable impact on reducing security risk, and should therefore be considered a success.

Example: Increasing HTTPS usage

HTTPS adoption on the web has increased dramatically in the last decade, driven by the concerted efforts of the Google Chrome team, Let's Encrypt, and other organizations. HTTPS provides important confidentiality and integrity guarantees for users and websites, and is critical to the web ecosystem's success—it's now required as part of HTTP/2.

To promote HTTPS use across the web, we conducted extensive research to build a strategy, contacted site owners using a variety of outreach channels, and set up

powerful incentives for them to migrate. Long-term, ecosystem-wide change requires a thoughtful strategy and significant planning. We used a data-driven approach to determine the best way to reach each stakeholder group:

- We gathered data about current HTTPS usage worldwide to select target regions.
- We surveyed end users to understand how they perceived the HTTPS UI in browsers.
- We measured site behavior to identify web platform features that could be restricted to HTTPS to protect user privacy.
- We used case studies to understand developer concerns about HTTPS and the types of tooling we could build to help.

This wasn't a one-time effort: we continued to monitor metrics and gather data over a multiyear period, adjusting our strategy when necessary. For example, as we gradually rolled out Chrome warnings for insecure pages over a period of years, we monitored user behavior telemetry to ensure that the UI changes didn't cause unexpected negative effects (for example, a dip in retention or engagement with the web).

Overcommunicating was key to success. Before each change, we used every available outreach channel: blogs, developer mailing lists, press, Chrome help forums, and relationships with partners. This approach maximized our reach so site owners weren't surprised that they were being pushed to move to HTTPS. We also tailored our outreach regionally—for example, devoting special attention to Japan when we realized that HTTPS adoption was lagging there because of slow uptake among top sites.

Ultimately, we focused on creating and emphasizing incentives to provide a business reason to migrate to HTTPS. Even security-minded developers had trouble convincing their organizations unless they could tie the migration to a business case. For example, enabling HTTPS allows websites to use web platform features like Service Worker, a background script that enables offline access, push notifications, and periodic background sync. Such features, which are restricted to HTTPS websites, improve performance and availability—and may have a direct impact on a business's bottom line. Organizations were more willing to devote resources to moving to HTTPS when they felt that the move was aligned with their business interests.

As shown in Figure 7-1, Chrome users across platforms now spend over 90% of their time on HTTPS sites, whereas previously this figure was as low as 70% for Chrome users on Windows and 37% for Chrome on Android. Countless people from many organizations—web browser vendors, certificate authorities, and web publishers—contributed to this increase. These organizations coordinated through standards bodies, research conferences, and open communication about the challenges and

successes that each faced. Chrome's role in this shift produced important lessons about contributing to ecosystem-wide change:

Understand the ecosystem before committing to a strategy.
: We based our strategy on quantitative and qualitative research, focusing on a broad set of stakeholders including web developers and end users in different countries and on different devices.

Overcommunicate to maximize reach.
: We used a variety of outreach channels to reach the widest range of stakeholders.

Tie security changes to business incentives.
: Organizational leaders were more amenable to HTTPS migration when they could see the business reasons.

Build an industry consensus.
: Multiple browser vendors and organizations supported the web's migration to HTTPS simultaneously; developers saw HTTPS as an industry-wide trend.

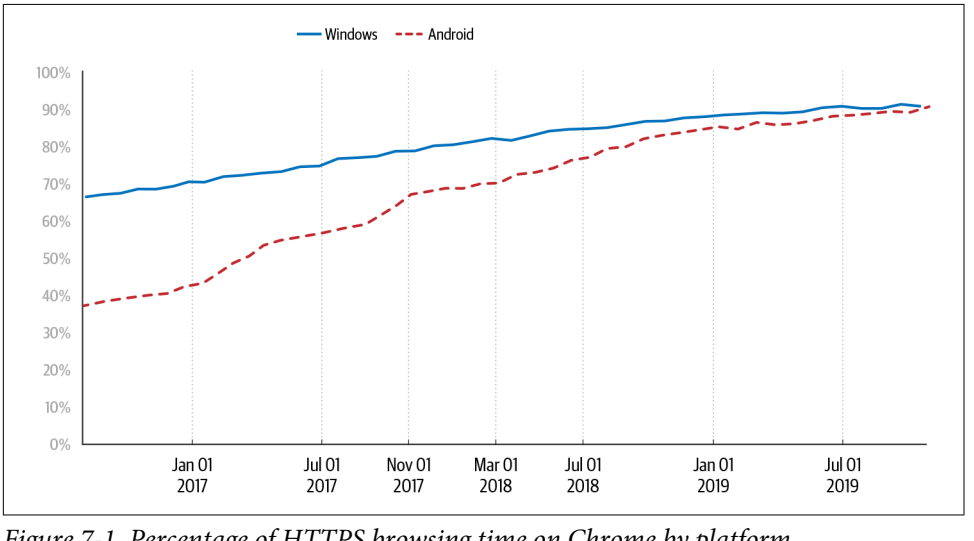

Figure 7-1. Percentage of HTTPS browsing time on Chrome by platform

Complications: When Plans Change

The best-laid plans of security and SRE often go awry. There are many reasons you may either need to accelerate a change or slow it down.

Often, you need to accelerate a change based upon external factors—typically, due to a vulnerability that's actively being exploited. In this case, you might want to speed up your rollout to patch your systems as quickly as possible. Be cautious: speeding up and breaking systems isn't necessarily better for the security and reliability of your

systems. Consider whether you can change the order of rollouts to cover certain higher-risk systems sooner, or otherwise remove attackers' access by rate limiting operations or taking a particularly critical system offline.

You may also decide to slow down a change. This approach is usually due to an issue with a patch, such as a higher than expected error rate or other rollout failures. If slowing down a change doesn't address the issue, or you can't roll out fully without negatively impacting your system or users, then rolling back, debugging the issue, and rolling out again is a painful but cautious approach. You may also be able to slow down a change based on updated business requirements—for example, changes to internal deadlines or delays in industry standards. (What do you mean, TLS 1.0 is still in use?!)

In the best-case scenario, your plans change before you start implementing them. In response, you just need to create new plans! Here are some potential reasons to change your plans, and corresponding tactics for doing so:

You might need to delay a change based on external factors.
: If you're not able to start patching as soon as the embargo lifts (see "Different Changes: Different Speeds, Different Timelines" on page 127), work with the vulnerability team to see if any other systems are in your situation, and if it's possible to change the timeline. Either way, make sure you have communications ready to inform affected users of your plan.

You might need to speed up a change based on a public announcement.
: For a vulnerability under embargo, you may have to wait for a public announcement before patching. Your timeline might change if the announcement leaks, if an exploit is made publicly available, or if the vulnerability is exploited in the wild. In this case, you'll want to start patching sooner rather than later. You should have an action plan at every stage for what to do if the embargo is broken.

You might not be severely impacted.
: If a vulnerability or change primarily affects public web-facing services, and your organization has a very limited number of such services, you probably don't need to rush to patch your entire infrastructure. Patch what's affected, and slow down the rate at which you apply patches to other areas of your system.

You might be dependent on external parties.
: Most organizations depend on third parties to distribute patched packages and images for vulnerabilities, or rely on software and hardware delivery as part of an infrastructure change. If a patched OS isn't available or the hardware you need is on backorder, there's likely not much you can do. You'll probably have to start your change later than originally intended.

Example: Growing Scope—Heartbleed

In December 2011, support for SSL/TLS's heartbeat feature was added to OpenSSL, along with an unrecognized bug that allowed a server or client to access 64 KB of another server or client's private memory. In April 2014, the bug was codiscovered by a Googler, Neel Mehta, and an engineer working for Codenomicon (a cybersecurity company); both reported it to the OpenSSL project. The OpenSSL team committed a code change to fix the bug and formulated a plan to publicly disclose it. In a move that surprised many in the security community, Codenomicon made a public announcement and launched the explanatory website *heartbleed.com*. This first use of a clever name and logo caused unexpectedly large media interest.

With early access to the patch, which was under embargo, and ahead of the planned public disclosure, Google infrastructure teams had already quietly patched a small number of key externally facing systems that directly handled TLS traffic. However, no other internal teams knew about the issue.

Once the bug became publicly known, exploits were developed quickly in frameworks such as Metasploit. Facing an accelerated timeline, many more Google teams now needed to patch their systems in a hurry. Google's security team used automated scanning to uncover additional vulnerable systems, and notified affected teams with instructions to patch and to track their progress. The memory disclosure meant that private keys could be leaked, which meant that a number of services needed key rotation. The security team notified affected teams and tracked their progress in a central spreadsheet.

Heartbleed illustrates a number of important lessons:

Plan for the worst-case scenario of the embargo breaking or being lifted early.
: While responsible disclosure is ideal, accidents (and cute logos that spur media interest) can happen. Do as much pre-work as possible, and move quickly to patch the most vulnerable systems regardless of disclosure agreements (which necessitate internal secrecy about embargoed information). If you can obtain a patch ahead of time, you may be able to deploy it before a public announcement. When that isn't possible, you should still be able to test and validate your patch to ensure a smooth rollout process.

Prepare for rapid deployment at scale.
: Use continuous builds to ensure you can recompile anytime, with a canary strategy to validate without destruction.

Regularly rotate your encryption keys and other secrets.

Key rotation is a best practice to limit the potential blast radius of a key compromise. Practice this operation regularly and confirm that your systems still work as intended; see Chapter 9 in the SRE workbook for details. By doing so, you ensure that swapping out any compromised keys won't be a Herculean effort.

Make sure you have a communication channel to your end users—both internal and external.

When a change fails or causes unexpected behavior, you need to be able to provide updates quickly.

Conclusion

Distinguishing between different kinds of security changes is critically important so that affected teams know what's expected of them and how much support you can offer.

Next time you're tasked with making a security change in your infrastructure, take a deep breath and create a plan. Start small or find volunteers willing to test the change. Have a feedback loop to understand what's not working for users, and make the change self-service. If plans change, don't panic—just don't be surprised, either.

Design strategies such as frequent rollouts, containerization, and microservices make both proactive improvements and emergency mitigation easier, while a layered approach makes for few and well-managed external surface areas. Thoughtful design and ongoing documentation, both with an eye toward change, keep your system healthy, make your team's workload more manageable, and—as you'll see in the next chapter—lead to greater resilience.