CHAPTER 5

Design for Least Privilege

By Oliver Barrett, Aaron Joyner, and Rory Ward with Guy Fischman and Betsy Beyer

The *principle of least privilege* says that users should have the minimum amount of access needed to accomplish a task, regardless of whether the access is from humans or systems. These restrictions are most effective when you add them at the beginning of the development lifecycle, during the design phase of new features. Unnecessary privilege leads to a growing surface area for possible mistakes, bugs, or compromise, and creates security and reliability risks that are expensive to contain or minimize in a running system.

In this chapter, we discuss how to classify access based on risk and examine best practices that enforce least privilege. A configuration distribution example demonstrates the tradeoffs these practices entail in real-world environments. After detailing a policy framework for authentication and authorization, we take a deep dive into advanced authorization controls that can mitigate both the risk of compromised accounts and the risk of mistakes made by on-call engineers. We conclude by acknowledging tradeoffs and tensions that might arise when designing for least privilege. In an ideal world, the people using a system are well-intentioned and execute their work tasks in a perfectly secure way, without error. Unfortunately, reality is quite different. Engineers may make mistakes, their accounts may be compromised, or they may be malicious. For these reasons, it's important to design systems for *least privilege*: systems should limit user access to the data and services required to do the task at hand.

Companies often want to assume their engineers have the best intentions, and rely on them to flawlessly execute Herculean tasks. This isn't a reasonable expectation. As a thought exercise, think about the damage you could inflict to your organization if you *wanted* to do something evil. What could you do? How would you do it? Would you be detected? Could you cover your tracks? Or, even if your intentions were good, what's the worst mistake you (or someone with equivalent access) could make? When debugging, responding to an outage, or performing emergency response, how many ad hoc, manual commands used by you or your coworkers are one typo or copy-paste fail away from causing or worsening an outage?

Because we can't rely on human perfection, we must assume that any possible bad action or outcome can happen. Therefore, we recommend designing the system to minimize or eliminate the impact of these bad actions.

Even if we generally trust the humans accessing our systems, we need to limit their privilege and the trust we place in their credentials. Things can and will go wrong. People will make mistakes, fat-finger commands, get compromised, and fall for phishing emails. Expecting perfection is an unrealistic assumption. In other words—to quote an SRE maxim—hope is not a strategy.

Concepts and Terminology

Before we dive into best practices for designing and operating an access control system, let's establish working definitions for some particular terms of art used in the industry and at Google.

Least Privilege

Least privilege is a broad concept that's well established in the security industry. The high-level best practices in this chapter can lay the foundations of a system that grants the least privilege necessary for any given task or action path. This goal applies to the humans, automated tasks, and individual machines that a distributed system comprises. The objective of least privilege should extend through all authentication and authorization layers of the system. In particular, our recommended approach rejects extending implicit authority to tooling (as illustrated in "Worked Example: Configuration Distribution" on page 74) and works to ensure that users don't have ambient authority—for example, the ability to log in as root—as much as practically possible.

Zero Trust Networking

The design principles we discuss begin with *zero trust networking*—the notion that a user's network location (being within the company's network) doesn't grant any privileged access. For example, plugging into a network port in a conference room does not grant more access than connecting from elsewhere on the internet. Instead, a system grants access based on a combination of user credentials and device credentials—what we know about the user and what we know about the device. Google has successfully implemented a large-scale zero trust networking model via its BeyondCorp program.

Zero Touch

The SRE organization at Google is working to build upon the concept of least privilege through automation, with the goal of moving to what we call *Zero Touch* interfaces. The specific goal of these interfaces—like Zero Touch Production (ZTP), described in Chapter 3, and Zero Touch Networking (ZTN)—is to make Google safer and reduce outages by removing direct human access to production roles. Instead, humans have indirect access to production through tooling and automation that make predictable and controlled changes to production infrastructure. This approach requires extensive automation, new safe APIs, and resilient multi-party approval systems.

Classifying Access Based on Risk

Any risk reduction strategy comes with tradeoffs. Reducing the risk introduced by human actors likely entails additional controls or engineering work, and can introduce tradeoffs to productivity; it may increase engineering time, process changes, operational work, or opportunity cost. You can help limit these costs by clearly scoping and prioritizing what you want to protect.

Not all data or actions are created equal, and the makeup of your access may differ dramatically depending on the nature of your system. Therefore, you shouldn't protect all access to the same degree. In order to apply the most appropriate controls and avoid an all-or-nothing mentality, you need to classify access based on impact, security risk, and/or criticality. For example, you likely need to handle access to different types of data (publicly available data versus company data versus user data versus cryptographic secrets) differently. Similarly, you likely need to treat administrative APIs that can delete data differently than service-specific read APIs.

Your classifications should be clearly defined, consistently applied, and broadly understood so people can design systems and services that "speak" that language. Your classification framework will vary based on the size and complexity of your system: you may need only two or three types that rely on ad hoc labeling, or you may need a robust and programmatic system for classifying parts of your system (API groupings, data types) in a central inventory. These classifications may apply to data stores, APIs, services, or other entities that users may access in the course of their work. Ensure that your framework can handle the most important entities within your systems.

Classifying Access Based on Risk | 63

Once you've established a foundation of classification, you should consider the controls in place for each. You need to consider several dimensions:

- Who should have access?
- How tightly should that access be controlled?
- What type of access does the user need (read/write)?
- What infrastructure controls are in place?

For example, as shown in Table 5-1, a company may need three classifications: *public*, *sensitive*, and *highly sensitive*. That company might categorize security controls as *low risk*, *medium risk*, or *high risk* by the level of damage the access can allow if granted inappropriately.

Table 5-1. Example access classifications based on risk

	Description	Read access	Write access	Infrastructure access[a]
Public	Open to anyone in the company	Low risk	Low risk	High risk
Sensitive	Limited to groups with business purpose	Medium/high risk	Medium risk	High risk
Highly sensitive	No permanent access	High risk	High risk	High risk

[a] Administrative ability to bypass normal access controls. For example, the ability to reduce logging levels, change encryption requirements, gain direct SSH access to a machine, restart and reconfigure service options, or otherwise affect the availability of the service(s).

Intersection of Reliability and Security: Permissions

From a reliability perspective, you might start with infrastructure controls, as you want to make sure unauthorized actors can't shut down jobs, change ACLs, and misconfigure services. However, keep in mind that from a security perspective, reading sensitive data can often be just as damaging: overly broad read permissions can lead to a mass data breach.

Your goal should be to construct an access framework from which you can apply appropriate controls with the right balance of productivity, security, and reliability. Least privilege should apply across all data access and actions. Working from this foundational framework, let's discuss how to design your systems with the principles and controls for least privilege.

Best Practices

When implementing a least privilege model, we recommend several best practices, detailed here.

Small Functional APIs

> Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features."
>
> —McIlroy, Pinson, and Tague (1978)[1]

As this quote conveys, Unix culture centers around small and simple tools that can be combined. Because modern distributed computing evolved from the single time-shared computing systems of the 1970s into planet-wide network-connected distributed systems, the authors' advice still rings true more than 40 years later. To adapt this quote to the current computing environment, one might say, "Make each API endpoint do one thing well." When building systems with an eye toward security and reliability, avoid open-ended interactive interfaces—instead, design around small functional APIs. This approach allows you to apply the classic security principle of least privilege and grant the minimum permissions necessary to perform a particular function.

What exactly do we mean by *API*? Every system has an API: it is simply the user interface the system presents. Some APIs are very large (such as the POSIX API[2] or the Windows API[3]), some are relatively small (such as memcached[4] and NATS[5]), and some are downright tiny (such as the World Clock API, TinyURL,[6] and the Google Fonts API[7]). When we talk about the API of a distributed system, we simply mean the sum of all the ways you can query or modify its internal state. API design has been

1 McIlroy, M.D., E.N. Pinson, and B.A. Tague. 1978. "UNIX Time-Sharing System: Foreword." *The Bell System Technical Journal* 57(6): 1899–1904. doi:10.1002/j.1538-7305.1978.tb02135.x.

2 POSIX stands for Portable Operating System Interface, the IEEE standardized interface provided by most Unix variants. For a general overview, see Wikipedia.

3 The Windows API includes the familiar graphical elements, as well as the programming interfaces like DirectX, COM, etc.

4 Memcached is a high-performance, distributed memory object caching system.

5 NATS is an example of a basic API built on top of a text protocol, as opposed to a complex RPC interface like gRPC.

6 The TinyURL.com API isn't well documented, but it is essentially a single GET URL that returns the shortened URL as the body of the response. This is a rare example of a mutable service with a tiny API.

7 The Fonts API simply lists the fonts currently available. It has exactly one endpoint.

well covered in computing literature;[8] this chapter focuses on how you can design and safely maintain secure systems by exposing API endpoints with few well-defined primitives. For example, the input you evaluate might be CRUD (Create, Read, Update, and Delete) operations on a unique ID, rather than an API that accepts a programming language.

In addition to the user-facing API, pay careful attention to the administrative API. The administrative API is equally important (arguably, more important) to the reliability and security of your application. Typos and mistakes when using these APIs can result in catastrophic outages or expose huge amounts of data. As a result, administrative APIs are also some of the most attractive attack surfaces for malicious actors.

Administrative APIs are accessed only by internal users and tooling, so relative to user-facing APIs, they can be faster and easier to change. Still, after your internal users and tooling start building on any API, there will still be a cost to changing it, so we recommend carefully considering their design. Administrative APIs include the following:

- Setup/teardown APIs, such as those used to build, install, and update software or provision the container(s) it runs in
- Maintenance and emergency APIs, such as administrative access to delete corrupted user data or state, or to restart misbehaving processes

Does the size of an API matter when it comes to access and security? Consider a familiar example: the POSIX API, one of our previous examples of a very large API. This API is popular because it is flexible and familiar to many people. As a production machine management API, it is most often used for relatively well-defined tasks such as installing a software package, changing a configuration file, or restarting a daemon.

Users often perform traditional Unix[9] host setup and maintenance via an interactive OpenSSH session or with tools that script against the POSIX API. Both approaches expose the entire POSIX API to the caller. It can be difficult to constrain and audit a user's actions during that interactive session. This is especially true if the user is

8 A good starting point is Gamma, Erich et al. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley. See also Bloch, Joshua. 2006. "How to Design a Good API and Why It Matters." *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*: 506–507. doi:10.1145/1176617.1176622.

9 Although we use Unix hosts as an example, this pattern is not unique to Unix. Traditional Windows host setup and management follows a similar model, where the interactive exposure of the Windows API is typically via RDP instead of OpenSSH.

maliciously attempting to circumvent the controls, or if the connecting workstation is compromised.

Can't I Just Audit the Interactive Session?

Many simple approaches to auditing, such as capturing the commands executed by bash or wrapping an interactive session in `script(1)`, may seem sufficient and quite comprehensive. However, these basic logging options for interactive sessions are similar to locks on the doors of a house: they only keep the honest people honest. Unfortunately, an attacker who is aware of their existence can easily bypass most of these types of basic session auditing.

As a simple example, an attacker might bypass bash command history logging by opening an editor like vim and executing commands from within the interactive session (e.g., `:!/bin/evilcmd`). This simple attack also complicates your ability to inspect the typescript log output by `script(1)`, because the output will be muddied by the visual control characters that vim and ncurses use to display the visual editing environment. Stronger mechanisms, like capturing some or all syscalls to the kernel's audit framework, are available, but it's worth understanding the shortcomings of naive approaches, and the difficulty in implementing stronger approaches.

You can use various mechanisms to limit the permissions granted to the user via the POSIX API,[10] but that necessity is a fundamental shortcoming of exposing a very large API. Instead, it's better to reduce and decompose this large administrative API into smaller pieces. You can then follow the principle of least privilege to grant permission only to the specific action(s) required by any particular caller.

The exposed POSIX API should not be confused with the OpenSSH API. It is possible to leverage the OpenSSH protocol and its authentication, authorization, and auditing (AAA) controls without exposing the entire POSIX API; for example, using git-shell.

Breakglass

Named after fire alarm pulls that instruct the user to "break glass in case of emergency," a *breakglass mechanism* provides access to your system in an emergency situation and bypasses your authorization system completely. This can be useful for

10 Popular options include running as an unprivileged user and then encoding allowed commands via `sudo`, granting only the necessary `capabilities(7)`, or using a framework like SELinux.

recovering from unforeseen circumstances. For more context, see "Graceful Failure and Breakglass Mechanisms" on page 74 and "Diagnosing Access Denials" on page 73.

Auditing

Auditing primarily serves to detect incorrect authorization usage. This can include a malicious system operator abusing their powers, a compromise of a user's credentials by an external actor, or rogue software taking unexpected actions against another system. Your ability to audit and meaningfully detect the signal in the noise is largely dependent on the design of the systems you're auditing:

- How granular is the access control decision being made or bypassed? (*What? Where?*)
- How clearly can you capture the metadata associated with the request? (*Who? When? Why?*)

The following tips will help in crafting a sound auditing strategy. Ultimately, your success will also depend on the culture associated with auditing.

Collecting good audit logs

Using small functioning APIs (as discussed in "Small Functional APIs" on page 65) provides the largest single advantage to your auditing ability. The most useful audit logs capture a granular series of actions, such as "pushed a config with cryptographic hash 123DEAD...BEEF456" or "executed <x> command." Thinking about how to display and justify your administrative actions to your customers can also help make your audit logs more descriptive, and thus more useful internally. Granular audit log information enables strong assertions about what actions the user did or did not take, but be sure to focus on capturing the *useful* parts of the actions.

Exceptional circumstances require exceptional access, which requires a strong culture of auditing. If you discover that the existing small functional API surfaces are insufficient to recover the system, you have two options:

- Provide breakglass functionality that allows a user to open an interactive session to the powerful and flexible API.
- Allow the user to have direct access to credentials in a way that precludes reasonable auditing of their usage.

In either of these scenarios, you may be unable to build a granular audit trail. Logging that the user opened an interactive session to a large API does not meaningfully tell you what they did. A motivated and knowledgeable insider can trivially bypass many solutions that capture session logs of interactive sessions, such as recording bash command history. Even if you can capture a full session transcript, effectively auditing it may be quite difficult: visual applications using ncurses need to be replayed to be human-readable, and features such as SSH multiplexing can further complicate capturing and understanding the interleaved state.

The antidote to overly broad APIs and/or frequent breakglass use is to foster a culture that values careful auditing. This is critical both for reliability reasons and security reasons, and you can use both motivations to appeal to the responsible parties. Two pairs of eyes help avoid typos and mistakes, and you should always safeguard against unilateral access to user data.

Ultimately, the teams building the administrative APIs and automation need to design them in a way that facilitates auditing. Anyone who regularly accesses production systems should be incentivized to solve these problems collaboratively and to understand the value of a good audit log. Without cultural reinforcement, audits can become rubber stamps, and breakglass use can become an everyday occurrence, losing its sense of importance or urgency. Culture is the key to ensuring that teams choose, build, and use systems in ways that support auditing; that these events occur only rarely; and that audit events receive the scrutiny they deserve.

Choosing an auditor

Once you have collected a good audit log, you need to choose the right person to inspect the (hopefully rare) recorded events. An auditor needs to have both the right context and the right objective.

When it comes to context, an auditor needs to know what a given action does, and ideally why the actor needed to perform that action. The auditor will therefore usually be a teammate, a manager, or someone familiar with the workflows that require that action. You'll need to strike a balance between sufficient context and objectivity: while an internal reviewer might have a close personal relationship with the person who generated the audit event and/or want the organization to succeed, an external private auditor may want to continue to be hired by an organization.

Choosing an auditor with the right objective depends on the purpose of the audit. At Google, we perform two broad categories of auditing:

- Audits to ensure best practices are being followed
- Audits to identify security breaches

Generally speaking, "best practice" audits support our reliability objectives. For example, an SRE team might choose to audit breakglass events from the last week's on-call shift during a weekly team meeting. This practice provides a cultural peer pressure to use and improve smaller service administrative APIs, rather than using a breakglass mechanism to access a more flexible emergency-use API. Widely scoped breakglass access often bypasses some or all safety checks, exposing the service to a higher potential for human error.

Google typically distributes breakglass reviews down to the team level, where we can leverage the social norming that accompanies team review. Peers performing the review have context that enables them to spot even very well-disguised actions, which is key to preventing internal abuses and thwarting malicious insiders. For example, a peer is well equipped to notice if a coworker repeatedly uses a breakglass action to access an unusual resource that they likely don't actually need. This type of team review also helps identify shortcomings in administrative APIs. When breakglass access is required for a specific task, it often signals a need to provide a safer or more secure way to perform that task as part of the normal API. You can read more about this subject in Chapter 21.

At Google we tend to centralize the second type of audit, as identifying external security breaches benefits from a broad view of the organization. An advanced attacker may compromise one team, and then use that access to compromise another team, service, or role. Each individual team may not notice a couple of anomalous actions, and doesn't have the cross-team view to connect the dots between different sets of actions.

A central auditing team may also be equipped to build extra signaling and add code for additional audit events that aren't widely known. These types of tripwires can be especially useful in early detection, but you may not want to share their implementation details widely. You may also need to work with other departments in your organization (such as Legal and HR), to ensure that auditing mechanisms are appropriate, properly scoped, and documented.

We at Google associate structured data with audit log events using *structured justification*. When an event that generates an audit log occurs, we can associate it with a structured reference such as a bug number, ticket number, or customer case number. Doing so allows us to build programmatic checks of the audit logs. For example, if support personnel look at a customer's payment details or other sensitive data, they can associate that data to a particular customer case. Therefore, we can ensure that the observed data belongs to the customer that opened the case. It would be much harder to automate log verification if we relied upon free-text fields. Structured justification has been key to scaling our auditing efforts—it provides a centralized auditing team context that's critical to effective auditing and analysis.

Testing and Least Privilege

Proper testing is a fundamental property of any well designed system. Testing has two important dimensions with regard to least privilege:

- Testing *of* least privilege, to ensure that access is properly granted only to necessary resources
- Testing *with* least privilege, to ensure that the infrastructure for testing has only the access it needs

Testing of least privilege

In the context of least privilege, you need to be able to test that well-defined user profiles (i.e., data analyst, customer support, SRE) have enough privileges to perform their role, but no more.

Your infrastructure should let you do the following:

- Describe what a specific user profile needs to be able to do in their job role. This defines the minimal access (APIs and data) and the type of access (read or write, permanent or temporary) they need for their role.
- Describe a set of scenarios in which the user profile attempts an action on your system (i.e., read, bulk read, write, delete, bulk delete, administer) and an expected result/impact on your system.
- Run these scenarios and compare the actual result/impact against the expected result/impact.

Ideally, to prevent adverse effects on production systems, these are tests that run before code or ACL changes. If test coverage is incomplete, you can mitigate overly broad access via monitoring of access and alerting systems.

Testing with least privilege

Tests should allow you to verify the expected read/write behavior without putting service reliability, sensitive data, or other critical assets at risk. However, if you don't have proper test infrastructure in place—infrastructure that accounts for varied environments, clients, credentials, data sets, etc.—tests that need to read/write data or mutate service state can be risky.

Consider the example of pushing a configuration file to production, which we'll return to in the next section. As your first step in designing a testing strategy for this configuration push, you should provide a separate environment keyed with its own credentials. This setup ensures that a mistake in writing or executing a test won't impact production—for example, by overwriting production data or bringing down a production service.

Alternately, let’s say you’re developing a keyboard app that allows people to post memes with one click. You want to analyze users’ behavior and history so you can automatically recommend memes. Lacking a proper test infrastructure, you instead need to give data analysts read/write access to an entire set of raw user data in production to perform analysis and testing.

Proper testing methodology should consider ways to restrict user access and decrease risk, but still allow the data analysts to perform the tests they need to do their job. Do they need write access? Can you use anonymized data sets for the tasks they need to perform? Can you use test accounts? Can you operate in a test environment with anonymized data? If this access is compromised, what data is exposed?

Security and Reliability Tradeoff: Test Environments

Instead of building out a test infrastructure that faithfully mirrors production, time and cost considerations may tempt you to use special test accounts in production so changes don’t impact real users. This strategy may work in some situations, but can muddy the waters when it comes to your auditing and ACLs.

You can approach test infrastructure by starting small—don’t let perfect be the enemy of good. Start by thinking about ways you can most easily

- Separate environments and credentials
- Limit the types of access
- Limit the exposure of data

Initially, perhaps you can stand up short-lived tests on a cloud platform instead of building out an entire test infrastructure stack. Some employees may need only read or temporary access. In some cases, you may also be able to use representative or anonymized data sets.

While these testing best practices sound great in theory, at this point, you may be getting overwhelmed by the potential cost of building out a proper test infrastructure. Getting this right isn’t cheap. However, consider the cost of *not* having a proper test infrastructure: Can you be certain that every test of critical operations won’t bring down production? Can you live with data analysts having otherwise avoidable privileges to access sensitive data? Are you relying on perfect humans with perfect tests that are perfectly executed?

It’s important to conduct a proper cost–benefit analysis for your specific situation. It may not make sense to initially build the “ideal” solution. However, make sure you build a framework people will use. People need to perform testing. If you don't provide an adequate testing framework, they'll test in production, circumventing the controls you put in place.

Diagnosing Access Denials

In a complex system, where least privilege is enforced and trust must be earned by the client via a third factor, multi-party authorization, or another mechanism (see "Advanced Controls" on page 81), policy is enforced at multiple levels and at a fine granularity. As a result, policy denials can also happen in complex ways.

Consider the case in which a sane security policy is being enforced, and your authorization system denies access. One of three possible outcomes might occur:

- The client was correctly denied and your system behaved appropriately. Least privilege has been enforced, and all is well.
- The client was correctly denied, but can use an advanced control (such as multi-party authorization) to obtain temporary access.
- The client believes they were incorrectly denied, and potentially files a support ticket with your security policy team. For example, this might happen if the client was recently removed from an authorized group, or if the policy changed in a subtle or perhaps incorrect way.

In all cases, the caller is blind to the reason for denial. But could the system perhaps provide the client with more information? Depending on the caller's level of privilege, it can.

If the client has no or very limited privileges, the denial should remain blind—you probably don't want to expose information beyond a 403 Access Denied error code (or its equivalent), because details about the reasons for a denial could be exploited to gain information about a system and even to find a way to gain access. However, if the caller has certain minimal privileges, you can provide a token associated with the denial. The caller can use that token to invoke an advanced control to obtain temporary access, or provide the token to the security policy team through a support channel so they can use it to diagnose the problem. For a more privileged caller, you can provide a token associated with the denial *and* some remediation information. The caller can then attempt to self-remediate before invoking the support channel. For example, the caller might learn that access was denied because they need to be a member of a specific group, and they can then request access to that group.

There will always be tension between how much remediation information to expose and how much support overload the security policy team can handle. However, if you expose too much information, clients may be able to reengineer the policy from the denial information, making it easier for a malicious actor to craft a request that uses the policy in an unintended way. With this in mind, we recommend that in the early stages of implementing a zero trust model, you use tokens and have all clients invoke the support channel.

Graceful Failure and Breakglass Mechanisms

Ideally, you'll always be dealing with a working authorization system enforcing a sane policy. But in reality, you might run into a scenario that results in large-scale incorrect denials of access (perhaps due to a bad system update). In response, you need to be able to circumvent your authorization system via a breakglass mechanism so you can fix it.

When employing a breakglass mechanism, consider the following guidelines:

- The ability to use a breakglass mechanism should be highly restricted. In general, it should be available only to your SRE team, which is responsible for the operational SLA of your system.
- The breakglass mechanism for zero trust networking should be available only from specific locations. These locations are your *panic rooms*, specific locations with additional physical access controls to offset the increased trust placed in their connectivity. (The careful reader will notice that the fallback mechanism for zero trust networking, a strategy of distrusting network location, is…trusting network location—but with additional physical access controls.)
- All uses of a breakglass mechanism should be closely monitored.
- The breakglass mechanism should be tested regularly by the team(s) responsible for production services, to make sure it functions when you need it.

When the breakglass mechanism has successfully been utilized so that users regain access, your SREs and security policy team can further diagnose and resolve the underlying problem. Chapters 8 and 9 discuss relevant strategies.

Worked Example: Configuration Distribution

Let's turn to a real-world example. Distributing a configuration file to a set of web servers is an interesting design problem, and it can be practically implemented with a small functional API. The best practices for managing a configuration file are to:

1. Store the configuration file in a version control system.
2. Code review changes to the file.
3. Automate the distribution to a canary set first, health check the canaries, and then continue health checking all hosts as you gradually push the file to the fleet

of web servers.[11] This step requires granting the automation access to update the configuration file remotely.

There are many approaches to exposing a small API, each tailored to the function of updating the configuration of your web servers. Table 5-2 summarizes a few APIs you may consider, and their tradeoffs. The sections that follow explain each tactic in more depth.

Table 5-2. APIs that update web server configuration and their tradeoffs

	POSIX API via OpenSSH	Software update API	Custom OpenSSH ForceCommand	Custom HTTP receiver
API surface	Large	Various	Small	Small
Preexisting[a]	Likely	Yes	Unlikely	Unlikely
Complexity	High	High	Low	Medium
Ability to scale	Moderate	Moderate, but reusable	Difficult	Moderate
Auditability	Poor	Good	Good	Good
Can express least privilege	Poor	Various	Good	Good

[a] This indicates how likely or unlikely it is that you already have this type of API as part of an existing web server deployment.

POSIX API via OpenSSH

You can allow the automation to connect to the web server host via OpenSSH, typically connecting as the local user the web server runs as. The automation can then write the configuration file and restart the web server process. This pattern is simple and common. It leverages an administrative API that likely already exists, and thus requires little additional code. Unfortunately, leveraging the large preexisting administrative API introduces several risks:

- The role running the automation can stop the web server permanently, start another binary in its place, read any data it has access to, etc.
- Bugs in the automation implicitly have enough access to cause a coordinated outage of all of the web servers.
- A compromise of the automation's credentials is equivalent to a compromise of all of the web servers.

11 *Canarying* a change is rolling it out to production slowly, beginning with a small set of production endpoints. Like a canary in a coal mine, it provides warning signals if something goes wrong. See Chapter 27 in the SRE book for more.

Software Update API

You can distribute the config as a packaged software update, using the same mechanism you use to update the web server binary. There are many ways to package and trigger binary updates, using APIs of varying sizes. A simple example is a Debian package (*.deb*) pulled from a central repository by a periodic `apt-get` called from `cron`. You might build a more complex example using one of the patterns discussed in the following sections to trigger the update (instead of `cron`), which you could then reuse for both the configuration and the binary. As you evolve your binary distribution mechanism to add safety and security, the benefits accrue to your configuration, because both use the same infrastructure. Any work done to centrally orchestrate a canary process, coordinate health checking, or provide signatures/provenance/auditing similarly pays dividends for both of these artifacts.

Sometimes the needs of binary and configuration update systems don't align. For example, you might distribute an IP deny list in your configuration that needs to converge as quickly as is safely practical, while also building your web server binary into a container. In this case, building, standing up, and tearing down a new container at the same rate you want to distribute configuration updates may be too expensive or disruptive. Conflicting requirements of this type may necessitate two distribution mechanisms: one for binaries, another for configuration updates.

For many more thoughts on this pattern, see Chapter 9.

Custom OpenSSH ForceCommand

You can write a short script to perform these steps:

1. Receive the configuration from `STDIN`.
2. Sanity check the configuration.
3. Restart the web server to update the configuration.

You can then expose this command via OpenSSH by tying particular entries in an *authorized_keys* file with the `ForceCommand` option.[12] This strategy presents a very small API to the caller, which can connect via the battle-hardened OpenSSH protocol, where the only available action is to provide a copy of the configuration file.

12 `ForceCommand` is a configuration option to constrain a particular authorized identity to run only a single command. See the *sshd_config* manpage for more details.

Logging the file (or a hash of it[13]) reasonably captures the entire action of the session for later auditing.

You can implement as many of these unique key/`ForceCommand` combinations as you like, but this pattern can be hard to scale to many unique administrative actions. While you can build a text-based protocol on top of the OpenSSH API (such as git-shell), doing so starts down the path of building your own RPC mechanism. You're likely better off skipping to the end of that road by building on top of an existing framework such as gRPC or Thrift.

Custom HTTP Receiver (Sidecar)

You can write a small sidecar daemon—much like the `ForceCommand` solution, but using another AAA mechanism (e.g., gRPC with SSL, SPIFFE, or similar)—that accepts a config. This approach doesn't require modifying the serving binary and is very flexible, at the expense of introducing more code and another daemon to manage.

Custom HTTP Receiver (In-Process)

You could also modify the web server to expose an API to update its config directly, receiving the config and writing it to disk. This is one of the most flexible approaches, and bears a strong similarity to the way we manage configuration at Google, but it requires incorporating the code into the serving binary.

Tradeoffs

All but the large options in Table 5-2 provide opportunities to add security and safety to your automation. An attacker may still be able to compromise the web server's role by pushing an arbitrary config; however, choosing a smaller API means that the push mechanism won't implicitly allow that compromise.

You may be able to further design for least privilege by signing the config independently from the automation that pushes it. This strategy segments the trust between roles, guaranteeing that if the automation role pushing the configuration is compromised, the automation cannot also compromise the web server by sending a malicious config. To recall McIlroy, Pinson, and Tague 's advice, designing each piece

13 At scale, it may be impractical to log and store many duplicate copies of the file. Logging the hash allows you to correlate the config back to the revision control system, and detect unknown or unexpected configurations when auditing the log. As icing on the cake, and if space allows, you may wish to store rejected configurations to aid a later investigation. Ideally, all configs should be signed, indicating they came from the revision control system with a known hash, or rejected.

of the system to perform one task and perform that task well allows you to isolate trust.

The more granular control surface presented by a narrow API also allows you to add protection against bugs in the automation. In addition to requiring a signature to validate the config, you can require a bearer token[14] from a central rate limiter, created independently of your automation and targeted to each host in the rollout. You can very carefully unit test this general rate limiter; if the rate limiter is independently implemented, bugs affecting the rollout automation likely won't simultaneously affect it. An independent rate limiter is also conveniently reusable, as it can rate limit the config rollout of the web server, the binary rollout of the same server, reboots of the server, or any other task to which you wish to add a safety check.

A Policy Framework for Authentication and Authorization Decisions

> *authentication* [noun]: verifying the **identity** of a user or process
>
> *authorization* [noun]: evaluating if a request from a specific authenticated party **should be permitted**

The previous section advocates designing a narrow administrative API for your service, which allows you to grant the least amount of privilege possible to achieve a given action. Once that API exists, you must decide how to control access to it. Access control involves two important but distinct steps.

First, you must *authenticate* who is connecting. An authentication mechanism can range in complexity:

Simple: Accepting a username passed in a URL parameter
    Example: */service?username=admin*

More complex: Presenting a preshared secret
    Examples: WPA2-PSK, an HTTP cookie

Even more complex: Complex hybrid encryption and certificate schemes
    Examples: TLS 1.3, OAuth

Generally speaking, you should prefer to reuse an existing strong cryptographic authentication mechanism to identify the API's caller. The result of this authentication decision is commonly expressed as a username, a common name, a "principal," a

14 A *bearer token* is just a cryptographic signature, signed by the rate limiter, which can be presented to anyone with the rate limiter's public key. They can use that public key to validate that the rate limiter has approved this operation, during the validity window of the token.

"role," and so on. For purposes of this section, we use *role* to describe the interchangeable result of authentication.

Next, your code must make a decision: is this role *authorized* to perform the requested action? Your code may consider many attributes of the request, such as the following:

The specific action being requested
: Examples: URL, command being run, gRPC method

Arguments to the requested action
: Examples: URL parameters, `argv`, gRPC request

The source of the request
: Examples: IP address, client certificate metadata

Metadata about the authenticated role
: Examples: geographic location, legal jurisdiction, machine learning evaluation of risk

Server-side context
: Examples: rate of similar requests, available capacity

The rest of this section discusses several techniques that Google has found useful to improve upon, and scale up, the basic requirements of authentication and authorization decisions.

Using Advanced Authorization Controls

An access control list for a given resource is a familiar way to implement an authorization decision. The simplest ACL is a string matching the authenticated role, often combined with some notion of grouping—for example, a group of roles, such as "administrator," which expands to a larger list of roles, such as usernames. When the service evaluates the incoming request, it checks whether the authenticated role is a member of the ACL.

More complex authorization requirements, such as multi-factor authorization (MFA) or multi-party authorization (MPA), require more complex authorization code (for more on three-factor authorization and MPA, see "Advanced Controls" on page 81). In addition, some organizations may have to consider their particular regulatory or contractual requirements when designing authorization policies.

This code can be difficult to implement correctly, and its complexity can rapidly compound if many services each implement their own authorization logic. In our experience, it's helpful to separate the complexities of authorization decisions from core API design and business logic with frameworks like the AWS or GCP Identity &

Access Management (IAM) offerings. At Google, we also extensively use a variation of the GCP authorization framework for internal services.[15]

The security policy framework allows our code to make simple checks (such as "Can X access resource Y?") and evaluate those checks against an externally supplied policy. If we need to add more authorization controls to a particular action, we can simply change the relevant policy configuration file. This low overhead has tremendous functionality and velocity benefits.

Investing in a Widely Used Authorization Framework

You can enable authentication and authorization changes at scale by using a shared library to implement authorization decisions, and by using a consistent interface as widely as possible. Applying this classic modular software design advice in the security sphere yields surprising benefits. For example:

- You can add support for MFA or MPA to all service endpoints with a single library change.
- You can then implement this support for a small percentage of the actions or resources in all services with a single configuration change.
- You can improve reliability by requiring MPA for all actions that allow a potentially unsafe action, similar to a code review system. This process improvement can improve security against insider risk threats (for more about types of adversaries, see Chapter 2) by facilitating fast incident response (by bypassing revision control system and code review dependencies) without allowing broad unilateral access.

As your organization grows, standardization is your friend. A uniform authorization framework facilitates team mobility, as more people know how to code against and implement access controls with a common framework.

Avoiding Potential Pitfalls

Designing a complex authorization policy language is difficult. If the policy language is too simplistic, it won't achieve its goal, and you'll end up with authorization decisions spread across both the framework's policy and the primary codebase. If the policy language is too general, it can be very hard to reason about. To mitigate these concerns, you can apply standard software API design practices—in particular, an iterative design approach—but we recommend proceeding carefully to avoid both of these extremes.

15 Our internal variant supports our internal authentication primitives, avoids some circular dependency concerns, etc.

Carefully consider how the authorization policy is shipped to (or with) the binary. You may want to update the authorization policy, which will plausibly become one of the most security-sensitive pieces of configuration, independently of the binary. For additional discussion about configuration distribution, see the worked example in the previous section, Chapters 9 and 14 in this book, Chapter 8 in the SRE book, and Chapters 14 and 15 in the SRE workbook.

Application developers will need assistance with the policy decisions that will be encoded in this language. Even if you avoid the pitfalls described here and create an expressive and understandable policy language, more often than not it will still require collaboration between application developers implementing the administrative APIs and security engineers and SREs with domain-specific knowledge about your production environment, to craft the right balance between security and functionality.

Advanced Controls

While many authorization decisions are a binary yes/no, more flexibility is useful in some situations. Rather than requiring a strict yes/no, an escape valve of "maybe," paired with an additional check, can dramatically ease the pressures on a system. Many of the controls described here can be used either in isolation or in combination. Appropriate usage depends on the sensitivity of the data, the risk of the action, and existing business processes.

Multi-Party Authorization (MPA)

Involving another person is one classic way to ensure a proper access decision, fostering a culture of security and reliability (see Chapter 21). This strategy offers several benefits:

- *Preventing mistakes* or unintentional violations of policy that may lead to security or privacy issues.
- *Discouraging bad actors* from attempting to perform malicious changes. This includes both employees, who risk disciplinary action, and external attackers, who risk detection.
- *Increasing the cost of the attack* by requiring either compromise of at least one other person or a carefully constructed change that passes a peer review.
- *Auditing past actions* for incident response or postmortem analysis, assuming the reviews are recorded permanently and in a tamper-resistant fashion.
- *Providing customer comfort*. Your customers may be more comfortable using your service knowing that no single person can make a change by themselves.

MPA is often performed for a broad level of access—for example, by requiring approval to join a group that grants access to production resources, or the ability to act as a given role or credential. Broad MPA can serve as a valuable breakglass mechanism, to enable very unusual actions that you may not have specific workflows for. Where possible, you should try to provide more granular authorization, which can provide stronger guarantees of reliability and security. If the second party approves an action against a small functional API (see "Small Functional APIs" on page 65), they can have much more confidence in precisely what they are authorizing.

Potential Pitfalls

Make sure that approvers have enough context to make an informed decision. Does the information provided clearly identify who is doing what? You may need to specify config parameters and targets in order to reveal what a command is doing. This may be particularly important if you're displaying approval prompts on mobile devices and have limited screen space for details.

Social pressure around approvals may also lead to bad decisions. For example, an engineer might not feel comfortable rejecting a suspicious request if its issued by a manager, a senior engineer, or someone standing over their desk. To mitigate these pressures, you can provide the option to escalate approvals to a security or investigations team after the fact. Or, you might have a policy that all (or a percentage) of a certain type of approval are independently audited.

Before building a multi-party authorization system, make sure the technology and social dynamics allow someone to say no. Otherwise, the system is of little value.

Three-Factor Authorization (3FA)

In a large organization, MPA often has one key weakness that can be exploited by a determined and persistent attacker: all of the "multiple parties" use the same centrally managed workstations. The more homogeneous the fleet of workstations, the more likely it is that an attacker who can compromise one workstation can compromise several or even all of them.

A classic method to harden the fleet of workstations against attack is for users to maintain two completely separate workstations: one for general use, such as browsing the web and sending/receiving emails, and another more trusted workstation for communicating with the production environment. In our experience, users ultimately want similar sets of features and capabilities from those workstations, and maintaining two sets of workstation infrastructure for the limited set of users whose credentials require this increased level of protection is both expensive and difficult to sustain over time. Once this issue is no longer top of mind for management, people are less incentivized to maintain the infrastructure.

Mitigating the risk that a single compromised platform can undermine all authorization requires the following:

- Maintaining at least two platforms
- The ability to approve requests on two platforms
- (Preferably) The ability to harden at least one platform

Considering these requirements, another option is to require authorization from a hardened mobile platform for certain very risky operations. For simplicity and convenience, you can only allow RPCs to be originated from fully managed desktop workstations, and then require three-factor authorization from the mobile platform. When a production service receives a sensitive RPC, the policy framework (described in "A Policy Framework for Authentication and Authorization Decisions" on page 78) requires cryptographically signed approval from the separate 3FA service. That service then indicates that it sent the RPC to the mobile device, it was shown to the originating user, and they acknowledged the request.

Hardening mobile platforms is somewhat easier than hardening general-purpose workstations. We've found that users are generally more tolerant of certain security restrictions on mobile devices, such as additional network monitoring, allowing only a subset of apps, and connecting through a limited number of HTTP endpoints. These policies are also quite easy to achieve with modern mobile platforms.

Once you have a hardened mobile platform on which to display the proposed production change, you have to get the request to that platform and display it to the user. At Google, we reuse the infrastructure that delivers notifications to Android phones to authorize and report Google login attempts to our users. If you have the luxury of a similar hardened piece of infrastructure lying around, it might be useful to extend it to support this use case, but failing that, a basic web-based solution is relatively easy to create. The core of a 3FA system is a simple RPC service that receives the request to be authorized and exposes the request for authorization by the trusted client. The user requesting the 3FA-protected RPC visits the web URL of the 3FA service from their mobile device, and is presented with the request for approval.

It is important to distinguish what threats MPA and 3FA protect against, so you can decide on a consistent policy about when to apply them. MPA protects against unilateral insider risk as well as against compromise of an individual workstation (by requiring a second internal approval). 3FA protects against broad compromise of internal workstations, but does not provide any protection against insider threats when used in isolation. Requiring 3FA from the originator and simple web-based

MPA from a second party can provide a very strong defense against the combination of most of these threats, with relatively little organizational overhead.

3FA Is Not 2FA (or MFA)

Two-factor authentication (2FA) is a well-discussed subset of multi-factor authentication. It is specifically the attempt to combine "something you know" (a password) with "something you have" (an application or hardware token that produces cryptographic proof of presence), to form a strong authentication decision. For more on 2FA, see the case study "Example: Strong second-factor authentication using FIDO security keys" on page 133.

The key difference is that 3FA is attempting to provide stronger *authorization* of a specific request, not stronger *authentication* of a specific user. While we acknowledge that the *3* in *3FA* is a bit of a misnomer (it's approval from a second platform, not a third platform), it's a helpful shorthand that your "3FA device" is the mobile device that adds additional authorization for some requests beyond your first two factors.

Business Justifications

As mentioned in "Choosing an auditor" on page 69, you can enforce authorization by tying access to a structured business justification, such as a bug, incident, ticket, case ID, or assigned account. But building the validation logic may require additional work, and may also require process changes for the people staffing on-call or customer service.

As an example, consider a customer service workflow. In an anti-pattern sometimes found in small or immature organizations, a basic and nascent system may give customer service representatives access to all customer records, either for efficiency reasons or because controls don't exist. A better option would be to block access by default, and to only allow access to specific data when you can verify the business need. This approach may be a gradient of controls implemented over time. For example, it may start by only allowing access to customer service representatives assigned an open ticket. Over time, you can improve the system to only allow access to specific customers, and specific data for those customers, in a time-bound fashion, with customer approval.

When properly configured, this strategy can provide a strong authorization guarantee that access was appropriate and properly scoped. Structured justifications allow the automation to require that Ticket #12345 isn't a random number typed in to satisfy a simple regular expression check. Instead, the justification satisfies a set of access policies that balance operational business needs and system capabilities.

Security and Reliability Tradeoff: System Usability

If it's too difficult for engineers to find an approver every time they want to perform an action with elevated privileges, they'll develop potentially dangerous behaviors. For example, they might work around enforced best practices by providing generic business justifications (like "Team foo needed access") that don't fulfill a functional requirement. Patterns of generic justifications should trigger alarm signals in the auditing system.

Temporary Access

You can limit the risk of an authorization decision by granting temporary access to resources. This strategy can often be useful when fine-grained controls are not available for every action, but you still want to grant the least privilege possible with the available tooling.

You can grant temporary access in a structured and scheduled way (e.g., during on-call rotations, or via expiring group memberships) or in an on-demand fashion where users explicitly request access. You can combine temporary access with a request for multi-party authorization, a business justification, or another authorization control. Temporary access also creates a logical point for auditing, since you have clear logging about users who have access at any given time. It also provides data about where temporary access occurs so you can prioritize and reduce these requests over time.

Temporary access also reduces ambient authority. This is one reason that administrators favor `sudo` or "Run as Administrator" over operating as the Unix user *root* or Windows Administrator accounts—when you accidentally issue a command to delete all the data, the fewer permissions you have, the better!

Proxies

When fine-grained controls for backend services are not available, you can fall back to a heavily monitored and restricted proxy machine (or *bastion*). Only requests from these specified proxies are allowed to access sensitive services. This proxy can restrict dangerous actions, rate limit actions, and perform more advanced logging.

For example, you may need to perform an emergency rollback of a bad change. Given the infinite ways a bad change can happen, and the infinite ways it can be resolved, the steps required to perform a rollback may not be available in a predefined API or a tool. You can give a system administrator the flexibility to resolve an emergency, but introduce restrictions or additional controls that mitigate the risk. For example:

- Each command may need peer approval.
- An administrator may only connect to relevant machines.
- The computer that an administrator is using may not have access to the internet.
- You can enable more thorough logging.

As always, implementing any of these controls comes with an integration and operational cost, as discussed in the next section.

Tradeoffs and Tensions

Adopting a least privilege access model will definitely improve your organization's security posture. However, you must offset the benefits outlined in the previous sections against the potential cost of implementing that posture. This section considers some of those costs.

Increased Security Complexity

A highly granular security posture is a very powerful tool, but it's also complex and therefore challenging to manage. It is important to have a comprehensive set of tooling and infrastructure to help you define, manage, analyze, push, and debug your security policies. Otherwise, this complexity may become overwhelming. You should always aim to be able to answer these foundational questions: "Does a given user have access to a given service/piece of data?" and "For a given service/piece of data, who has access?"

Impact on Collaboration and Company Culture

While a strict model of least privilege is likely appropriate for sensitive data and services, a more relaxed approach in other areas can provide tangible benefits.

For example, providing software engineers with broad access to source code carries a certain amount of risk. However, this is counterbalanced by engineers being able to learn on the job according to their own curiosity and by contributing features and bug fixes outside of their normal roles when they can lend their attention and expertise. Less obviously, this transparency also makes it harder for an engineer to write inappropriate code that goes unnoticed.

Including source code and related artifacts in your data classification effort can help you form a principled approach for protecting sensitive assets while benefiting from visibility into less sensitive assets, which you can read more about in Chapter 21.

Quality Data and Systems That Impact Security

In a zero trust environment that is the underpinning of least privilege, every granular security decision depends on two things: the policy being enforced and the context of the request. *Context* is informed by a large set of data—some of it potentially dynamic—that can impact the decision. For example, the data might include the role of the user, the groups the user belongs to, the attributes of the client making the request, the training set fed into a machine learning model, or the sensitivity of the API being accessed. You should review the systems that produce this data to ensure that the quality of security-impacting data is as high as possible. Low-quality data will result in incorrect security decisions.

Impact on User Productivity

Your users need to be able to accomplish their workflows as efficiently as possible. The best security posture is one that your end users don't notice. However, introducing new three-factor and multi-party authorization steps may impinge on user productivity, especially if users must wait to be granted authorization. You can minimize user pain by making sure the new steps are easy to navigate. Similarly, end users need a simple way to make sense of access denials, either through self-service diagnosis or fast access to a support channel.

Impact on Developer Complexity

As the model for least privilege is adopted across your organization, developers must conform to it. The concepts and policies must be easily consumable by developers who aren't particularly security-savvy, so you should provide training materials and thoroughly document your APIs.[16] As they navigate the new requirements, give developers easy and fast access to security engineers for security reviews and general consulting. Deploying third-party software in this environment requires particular care, as you may need to wrap software in a layer that can enforce the security policy.

Conclusion

When designing a complex system, the least privilege model is the most secure way to ensure that clients have the ability to accomplish what they need to do, but no more. This is a powerful design paradigm to protect your systems and your data from malicious or accidental damage caused by known or unknown users. Google has spent significant time and effort implementing this model. Here are the key components:

16 See "A Policy Framework for Authentication and Authorization Decisions" on page 78.

- A comprehensive knowledge of the functionality of your system, so you can classify different parts according to the level of security risk each holds.
- Based on this classification, a partitioning of your system and access to your data to as fine a level as possible. Small functional APIs are a necessity for least privilege.
- An authentication system for validating users' credentials as they attempt to access your system.
- An authorization system that enforces a well-defined security policy that can be easily attached to your finely partitioned systems.
- A set of advanced controls for nuanced authorization. These can, for example, provide temporary, multi-factor, and multi-party approval.
- A set of operational requirements for your system to support these key concepts. At a minimum, your system needs the following:
  - — The ability to audit all access and to generate signals so you can identify threats and perform historical forensic analysis
  - — The means to reason about, define, test, and debug your security policy, and to provide end-user support for this policy
  - — The ability to provide a breakglass mechanism when your system does not behave as expected

Making all these components work in a way that is easy for users and developers to adopt, and that does not significantly impact their productivity, also requires an organizational commitment to making adoption of least privilege as seamless as possible. This commitment includes a focused security function that owns your security posture and interfaces with users and developers through security consulting, policy definition, threat detection, and support on security-related issues.

While this can be a large undertaking, we strongly believe it is a significant improvement over existing approaches to security posture enforcement.