Privilege Separation

6.1600 Course Staff

Fall 2023

The last chapter left off on a somewhat unsatisfying note: our software is bound to have bugs and those bugs are likely to be exploitable. Today, we will try to limit the impact that these bugs can have by *planning for compromise* and aiming to limit the damage that each component can cause if an attacker compromises it.

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

The philosophy that we will employ is called the *Principle of Least Privilege*. The idea is to give each component in the system the least privileges needed to do its job. By limiting the access that each component of our software has to do sensitive actions, we can prevent the compromise of our whole system when a single piece is attacked.

In real systems, it is often costly to implement least privilege in the strictest sense possible. You should think of the principle of least privileges as a difficult-to-achieve design goal, rather than a rule that every component of every system must satisfy.

To summarize our strategy:

> **Principle of Least Privilege:** Each component should have the smallest set of privileges necessary to do its job.

As a concrete example, a web server might have some networking code, it might talk to a database server, and it might use some cryptographic keys. One way to architect the system with least privilege would be to split the server into multiple distinct processes—one that receives network requests, one that interfaces with the database, and one that uses the cryptographic keys. These components then would speak to each other using narrow, restrictive APIs.

In some sense, a system that perfectly implements the principle of least privilege gives one with the "best security we could ask for," in the face of component compromise. However, most real systems do not completely implement the principle of least privilege for a number of reasons:

An orthogonal, but important, strategy is to reduce the number of privileges that a component needs to do its job.

Challenge: Splitting Boundaries. A large piece of software has many components that often have many points of interaction. To effectively implement least privilege, we need to find boundaries at which we can separate our software. A few common ways to partition a system or application are:

If we don't split up a large piece of software into components, the software just consists of one über-component. An attacker who breaks into this one component can hijack the entire system.

- **Isolating by user:** In an operating system, different users of the systems have different privileges. This way, if an attacker compromises one user's account it does not compromise the entire system.
- **Splitting by feature:** In large business applications, different features (e.g., search, user management, mail) run as isolated com-

ponents. This way, a bug in one component does not necessarily affect the entire system.

- **Splitting off buggy code:** The Firefox web browser uses special sandboxes to compartmentalizes bug-prone video codec code (Section 3). This way, if a malicious website is able to exploit the codec code, it is not so easy for the attacker to compromise the system.
- **Separating exposed versus internal code:** Google's front-end HTTP servers are isolated from the servers that run core application logic. This way, an attacker on the network is one step removed from the servers with access to Google's internal resources (databases, etc.).
- **Isolating sensitive data or keys:** Certificate authorities (CAs) use hardware security modules to isolate the cryptographic keys from all other code in the system. Laptops and servers sometimes have separate cryptographic co-processors for this purpose as well.

Challenge: Interface/API. If we chop our monolithic app into different domains, we also need to clearly define an API that the different domains can use to interact. This API must allow us to preserve the functionality of the system while making it difficult for an attacker that compromises one component from compromising an adjacent one.

Some strategies for interfacing between different components of a system might be

- remote procedure calls (RPCs) over a network,
- message queues,
- shared memory,
- shared database, or
- shared files or directories.

In designing interfaces between isolated components, we tend to worry about:

- **Functionality:** Does the isolation plan allow the application to work as it should?
- **Security:** Does the isolation strategy prevent the compromise of one component from affecting others?
- **Performance:** How much performance overhead does the isolation mechanism add?

- **Complexity:** How much code does the isolation mechanism require? If the compartmentalization strategy requires a large amount of code, this code might introduce new vulnerabilities.

1 *Example: Logging*

Most systems use logging to keep track of the actions that an application has performed. That way, if an attacker compromises the application, system administrators can look at the log to see what happened and how to mitigate it. For a logging system to be useful, it must be difficult for an attacker who compromises the application to erase the log.

It is very natural to separate out the log into a separate component: the app's functionality is almost entirely separate from the log, and the app's interface to the log can be very simple. The app should be allowed to append to the log and read from the log—the app should *not* be allowed to remove entries from the log. This way, compromise of the application or log server will not be enough to erase the log.

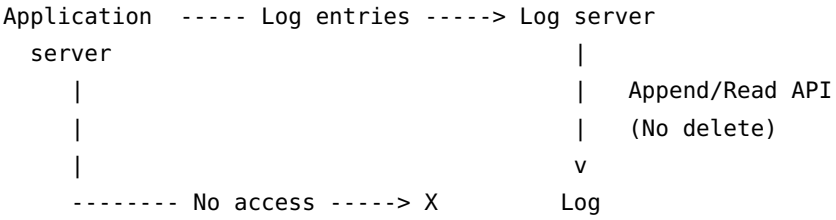

Figure 1: One way to architect a logging system. An attacker that compromises the application and/or log server may not be able to erase the logs.

2 *Example: Cryptography Keys*

Many applications use cryptographic keys for authentication: certificate authorities need to generate signed certificates for their clients, cryptocurrency wallets need to sign transactions, and WebAuthn requires an authenticator to sign a challenge from the relying party. In all of these applications, we worry a lot about an attacker stealing the application's secret signing key. Towards the goal of protecting cryptographic secrets, we often isolate the code that uses cryptographic keys into a separate software (or even hardware) component. In particular, we might create a cryptography component whose only API is `sign(msg)` and `get_public_key()`. In this way, even if an attacker compromises the application, it cannot easily extract the secret key.

This design does not completely protect us against the compromise of the application. In particular, an attacker who compromises our app can still call `sign()` as much as they like to sign any message

they like. At the same time, having the key isolated to this crypto component allows us to add checks inside the crypto component to, for example, reject messages of the wrong type. If our service is a certificate authority, we could set up our crypto component to verify that each message is an actual signature before signing it.

Although in many cases the things we are worried about an attacker asking for signatures of will pass this type check: for example, for a CA, this would still allow an attacker to generate a certificate for their malicious website.

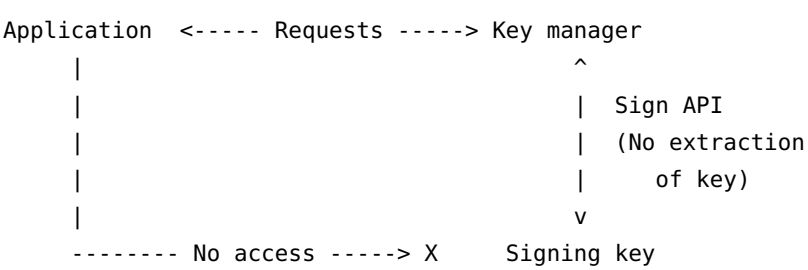

Figure 2: A common architecture for isolating cryptographic keys from an application.

This division also allows our crypto code to do some meaningful logging: imagine that the crypto module saves every signature it creates to a log. Compromise of the main app would allow an attacker to generate arbitrary signatures, but administrators could then see every signature that was generated during the compromise. This would be very helpful in recovery.

3 *Example: Media Codecs in Web Browsers*

Media codecs (e.g., for JPEG decoding) are notoriously complex and bug-prone: codec libraries have been at the source of many web-browser exploits. If we were able to isolate these codecs, we could make it less likely that a codec bug allows an attacker to access data other than the media file being decoded.

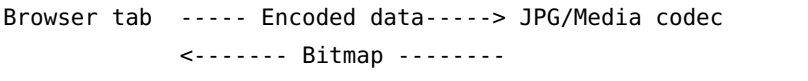

Figure 3: The Firefox browser isolates media codecs in a separate sandbox.

Isolating codecs may not always be as trivial as Fig. 3 makes it seem. In many cases, codecs require a sophisticated interface to the browser tab: the codec library may progressively decode videos as data arrives, for example. Even image decoders progressively decode images, improving resolution as more data arrives. Supporting this functionality without adding vulnerabilities or excessive latency requires careful API design.

Firefox isolates these risky codecs using the language-level isolation that the WebAssembly language provides.

4 *Example: Server for Network Time Protocol (NTP)*

Operating systems use the network-time protocol to fetch the current time from time servers on the Internet, and to update the current time to match. Setting the time requires root privilege on Unix-like systems, but NTP also requires network accesses; the networking code can be complicated and bug-prone. To prevent an attacker who finds a bug in these network protocols from gaining root privilege on our machine, many systems separate the two into a process that handles talking to the NTP server on the network and a privileged process that accepts the time from this other process and sets it. This way, an attacker who find a bug in the network code can only set the time—they cannot perform arbitrary actions as root. In addition, the time service may impose some policy on time changes (e.g,. that time can never go backwards).

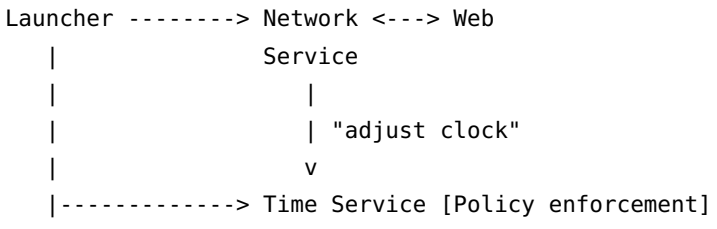

Figure 4: Modern operating systems implement a network-time protocol (NTP) client as separate processes. This way, an attacker who compromises the client over the network cannot arbitrarily corrupt the system time.

5 *Example: OpenSSH Server*

A secure shell (SSH) server has access to many sensitive resources: network port 22, a host secret key, the system's password file, and all users' data. When it starts, the SSH server runs a "monitor" process that listens for connections on port 22. When the monitor receives a connection, the first thing it does is to spawn a new per-connection worker process that communicates over the network. The worker client has no access to the host key or password database—the worker process can only ask the monitor for help in authenticating. The monitor-worker API supports a few operations:

- a *signing* operation, that instructs the monitor to sign a protocol transcript,
- a *password-authentication* operation, that instructs the monitor to check a password (this can be rate limited to prevent password guessing), and
- a *start-session* operation, that instructs the monitor to create a new process with a shell for the user.

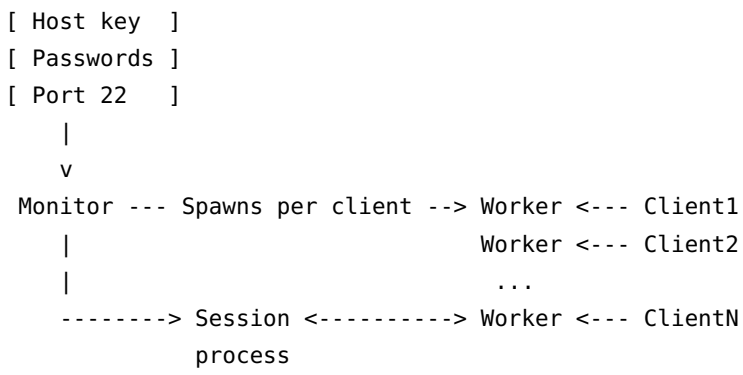

Figure 5: The OpenSSH server is split into multiple process to mitigate the compromise of the network-facing code.

While this architecture adds quite a bit of complexity to the OpenSSH server, it has paid off in terms of mitigating the impact of vulnerabilities in client-facing code.

6 Example: Web applications

Companies will often implement Web applications (e.g., a photo-sharing website) as a number of separate services, running on separate physical machines. Client connections come into a front-end server that terminates the TLS connection and proxies client data to a front-end application server. The front-end server then routes requests to one or more application services, each of which may have access to different databases. For example, the login service may have access to the password database, while the profile service may not.

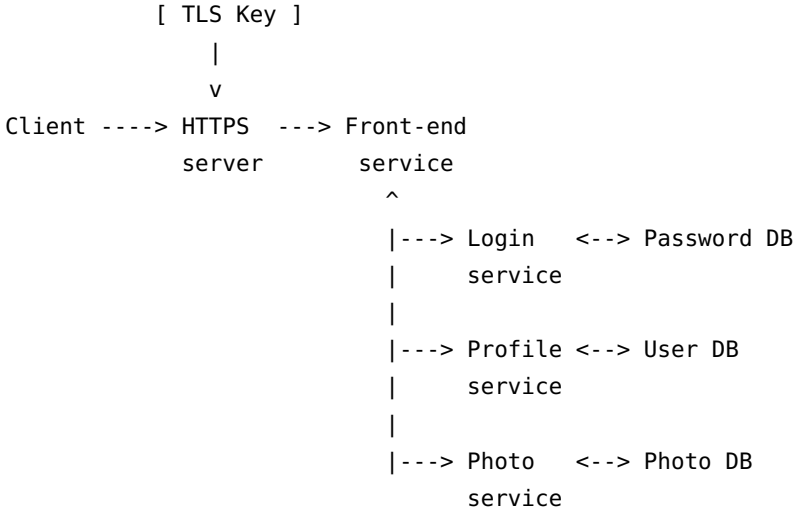

Figure 6: Large Web services tend to isolate different application features into different services, often on different physical machines. The system gates access to these services via minimal front-end client-facing servers.

7 *Example: Web client*

When you open a PDF attachment in Gmail , you might worry that the PDF could exploit some bug in your browser that could steal your sensitive Google data. To make this kind of attack more difficult, Gmail serves attachments and other suspect files from a separate domain ("origin')): `googleusercontent.com`. Code loaded from `googleusercontent.com` cannot access cookies or data for `google.com`. In this way, even if an attacker can somehow run JavaScript in your browser, it cannot easily steal your Google cookies.

8 *Example: Web browser*

Web browsers today are extraordinarily complicated pieces of software. The sensitive data that a browser is trying to protect are things, such as user cookies, cached data, browser history, and other user data. The browser may spawn new processes to handle rendering for each site from each distinct domain/origin. In this way, if an attacker from one origin can exploit a bug in the JavaScript engine, the attacker may still not be able to compromise sensitive user data from other domains/origins. GPU code, which is extremely complicated and bug-prone, may run in yet another process. Today, compromising a browser entirely often requires finding and exploiting a collection of bugs in multiple components.

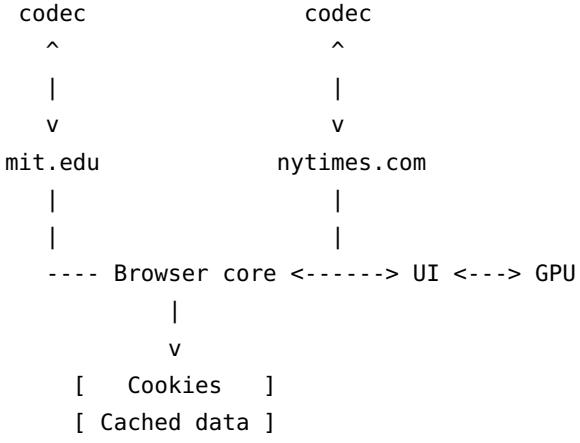

Figure 7: Web browsers may isolate the execution of each origin's code in a separate process. They further isolate complicated and bug-prone codecs and GPU code in separate processes.

9 *Example: Payment Systems*

Processing credit-card transactions in web applications is risky: if a vendor suffers a compromise, the credit-card network may fine

them or kick them off the network. To avoid ever having to handle credit-card data, most websites use an external payment-processing service that handles credit-card information. When the user makes a purchase, the vendor redirects the user to the payment-processing service, who collects the user's credit-card data. After payment, the payment-processing service redirects the user back to the vendor's website.

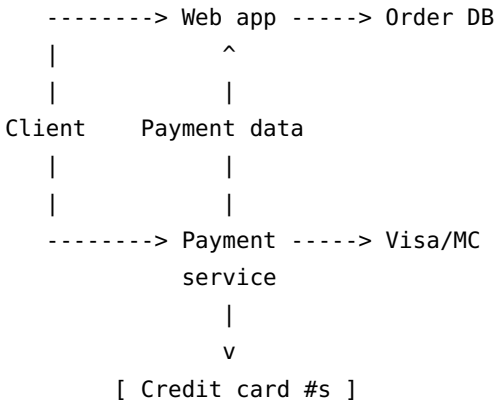

Figure 8: Online vendors often use a separate payment processor that handles the user's credit-card data.