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.

The philosophy that we will employ is called the Principle of Least Privilege. 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. To summarize our strategy:

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

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:

**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 on which we can separate our software. A few common ways to partition a system or application are:

- By user – standard in operating systems
- By feature – common on the server side of large business applications
- Splitting off buggy code – e.g., sandboxing video codecs in a web browser (Section 1.3)
- Exposed vs. Internal code – e.g., Google’s front-end HTTP servers are separate from core application logic
- Isolating sensitive keys – e.g., hardware security modules

**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 avoiding any bypasses of the permission system. As designers of this isolation, we must also be concerned with the performance overhead of the separation system and with the complexity overhead that the system adds—it is possible for the addition of this isolation to add new bugs and exploits.

1 Examples

1.1 Logging

It is often desirable in a system to have a reliable log of everything that happens: even if the system becomes compromised, this allows administrators to recover much more effectively by being able to see the actions that the attackers took. This log gives a clear isolation boundary: 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 likely not be allowed to remove entries from the log. This way, bugs outside of the logger will not be able to lead to the deletion of log entries.

1.2 Cryptography Keys

Many domains require the use of public-key cryptography to generate a signature over some data: 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, the worst-case scenario is an attacker with access to our private key. This also leads to a somewhat natural division of our application: we can create a cryptography component whose only API is $\text{sign}(\text{msg})$ and $\text{get_public_key}$. It should be impossible for an attacker to take advantage of a bug in the rest of our app to learn the private key.

However, this design does not guarantee everything that we might like: an attacker who compromises our app can still call $\text{sign}()$ as much as they like to sign any message they like. However, 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.

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.

1.3 Media Codecs in Web Browsers

Media codecs for things like JPEG decoding are notoriously bug-prone due to their complexity. This codec libraries have been the cause of many web browser exploits because of this. If we were able to isolate these codecs, we could prevent these bugs from allowing an attacker to access data other than the media file being decoded.

However, these codecs require a much more sophisticated interface than the simple logging and signing examples we have seen so far. Videos are progressively decoded as data arrives, for example, and even image decoders progressively decode images, improving resolution as more data arrives. Supporting this functionality without adding vulnerabilities or excessive latency presents many challenges.

Firefox in particular has a coherent plan for isolating these risky codecs by taking advantage of the language-level isolation provided by WebAssembly.

1.4 NTP Server

The Network Time Protocol, or NTP, is used by operating systems to fetch the current time from servers on the internet and update the current time to match. Setting the time requires root privilege on Unix-like systems, but NTP also requires network accesses that 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.

1.5 Payment Systems

Many applications employ privilege separation between their normal application code and the code that handles credit card numbers and other sensitive payment information. This separation will likely allow the app to initiate a charge for a certain amount. The interface here is again very important.