Software Trust

6.1600 Course Staff

Fall 2024

The central question of this chapter is:

> *How do we know whether a system is running the software that we expect it to be running?*

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.

This question comes up both when we are interacting with a machine in person (e.g., typing a passcode into our phone) or across a network (e.g., when sending sensitive data to a far-away server).

A separate question, which we will discuss in future chapters is: How do we ensure that the software itself is "good" or "bug-free?"

Threats to software integrity. There are a number of threats that might cause a machine to be running unexpected software:

- **Malware.** An adversary may install bad software onto the laptop, such as a keylogger.
- **User error.** A user may inadvertently install malware onto a machine.
- **Software supply-chain problems.** An adversary may inject malware into a real app's libraries, by tricking or coercing a developer.
- **Malicious updates.** An adversary may trick the software update process, converting a real piece of software into a malicious one.

A classic way to trick users into installing malware is to show them a warning (e.g., on a webpage) that says "Your computer is infected. Please download and install this anti-virus software."

The software supply chain. There are many steps that take place between the development of a piece of software and its use:

1. Developers write code. Their code may use many third-party libraries.
2. Compilers build and package the application.
3. The software vendor distributes binaries over the network.
4. Users download new software.
5. Users download software updates.
6. Users launch applications on their devices.
7. Running applications interact with remote servers running code.

1 Library Imports

1.1 Example: Python Imports

In Python, importing a library requires downloading a library using the `pip` package manager:

```
pip install requests
```

After that a user can import the package into their code like this:

```
import requests
```

Behind the scenes, the PyPi service maintains a database that maps package names (e.g., `requests`) to a piece of code. When you type `pip install requests`, the `pip` program fetches the code from the PyPi repository and installs it on your machine.

Benefits Benefits of this approach are:

- The centralized service makes it easy for users to identify packages.
- It is relatively easily for users to discover and download software updates.
- Developers do not need to run their own code-distribution service.

Downsides Downsides of this approach are:

- The centralized update service is a single point of failure: if an attacker is able to change the code in PyPi, it can infect a large number of machines at once.
- The end user has no idea who actually produced the library code. The user only is able to specify the package name.
- In Python, the naming scheme is ambiguous: if there is a public package and a private package both with the name `requests`, it's not clear when a user imports `requests`, which one the user wants to import.

1.2 *Example: Go Library Imports*

The Go programming language takes a slightly different approach to package management. In the Go programming language, a user imports a library/package by specifying the URL of the package's Git repository. For example, an import might look like this:

```
import "github.com/grpc/grpc-go"
```

When compiling the code, the user's PC will contact the server at the given URL over HTTPS (verifying the server certificate via TLS) and download the software bundle. On the other end, when a library developer wants to update their library, they do so by interacting with the hosting server via HTTPS and whatever authentication the server has set up—credentials, maybe two-factor authentication, etc.

This has some good features: the server name is explicit so there is no ambiguity about packages and the decentralized nature of specifying individual URLs avoids the necessity for a central server that attracts attacks. However, this requires trusting the server hosting the library to secure the update process and distribute software honestly.

1.3 More Explicit Trust: Code Signing

In each of these approaches, if an attacker can cause the user to download a bad package *without* compromising the package developer. In particular, if the attacker can compromise Github, it can cause Github to distribute malware to end users.

To prevent this attack, a library developer could sign their software using their private key and include the signature with their software package. To verify that a package is authentic, the application developer's PC can check that the signature is valid.

Of course, with any signature-based plan the mechanism for public key distribution is crucially important. In the software distribution case, the only reasonable plan is likely a Trust-on-First-Use based one which accepts the first public key it sees but verifies that future software updates use that same key. This protects against an adversary taking control of, for example, the application's Github repository after the end user installs the softare once. However, key management is hard, so this is not widely used in practice.

2 Building Binaries

In order to run software on our computer, it is necessary to convert the source code (which is, at least in principle, manually auditable) into a binary that is much more difficult to audit. Since compiling software is computation-heavy, most application developers typically compile their software and distribute the binary to their users. If an attacker compromises the build server (or is able to backdoor the compiler), then the attacker can cause users to execute bad code, even if the attacker does not compromise the application developer itself.

The XCodeGhost attack is an example of how an attacker can insert a backdoor in a build system and exploit it to distribute malware.

Reproducible Builds. One promising approach to the problem of ensuring that a binary is the faithful compilation of a piece of software is called "reproducible builds."

If a build process is reproducible, the function that turns a set of source-code files into a binary is a deterministic function: if two different people compile the same set of source-code files, they will get exactly the same binary—the two will be bit-for-bit identical. This allows anyone to *audit* a build: to check that a build server did its job correctly. In addition, having multiple independent parties build the same piece of software (and sign the result) can give an end user some assurance that the build server behaved correctly.

Implementing reproducible builds is not trivial. Traditional compilers introduced many sources of non-determinism—not necessarily for any particular reason, just for convenience. Creating reproducible builds requires eliminating all of these sources of non-determinism, even across multiple versions of the compiler.

As of today, the Go programming language now supports reproducible builds.

2.1 *Juggling multiple versions of a library*

Once a binary exists, the next step of the process is to distribute that software to user devices. Typically, there are many different versions of a piece of software around. When a user wants to install a piece of software, they typically need to specify which version of the software they want.

For example, in Python a user can specify a version of a package that they would like to install when they run `pip install`. If an attacker compromises the PyPi server, it can serve up any code it wants to a user asking for a particular version of a library.

In contrast, in the Go programming language, when a users imports a package, the `go get` software will store a hash of the downloaded code in a file called `go.sum`. If an attacker later compromises the server serving the package (e.g., Github), the `go get` command will refuse to install the package unless its code matches the stored hash value.

This is an example of "trust on first use" in the context of code installation.

3 *Installing & Updating Software*

Once a software developer finishes writing an application, it builds and distributes it. When a user installs an application—e.g., by downloading it from a website or fetching it using a package manager—how does the user know that it got the authentic version of the software? As usual in systems design, there are many possible strategies.

Application Developer Signs Package (Android Apps). One possible option is to have application developers sign the software that they produce. When application developers distribute their software, they attach a their signature to it. This way, it does not matter how a user obtains the software—a user can download an application bundle from any server and know that it came from the developer who owns the corresponding secret key.

When a user first installs a piece of software they need to somehow obtain the software developer's public key. Public-key distribution, as always, is messy: trust on first use is a common strategy.

Once the user has the software developer's public key, the user can easily verify that future updates to the software came from the same developer. (To do this, the user can just check signatures on the updates using the app-developer's public key.)

An important caveat is that signatures do not guarantee freshness: once signed, a package is always valid.

Repository Signs Packages. For systems with a central repository, another plan is for the repository to sign packages. This again allows the user to fetch the signed packages from untrustworthy sources—from a content-distribution network, for example.

In addition to signing the packages, the repository typically signs a timestamped manifest of the latest package versions. This allows a user to check that they are not only getting the right software but also that they are getting the most up-to-date software.

Many Linux package managers, such as `apt`, `pacman`, and `rpm`, use signatures to integrity-protect packages.

Third-Party Validator Signs Packages. Yet another option that does not require a single central repository is to have a trusted validator sign packages. This involves sending the source code and package to a third party, who will then perform some inspection of the package and, if it deems a package to be worthy, provide some signature over that package that verifies that the validator thinks the package is trustworthy.

A number of software platforms use this strategy for protecting binaries. On Android, there is no requirement to install apps from the Google Play Store, but Google provides a service that inspects packages and attaches these signatures if the package passes. Similarly, Windows uses a validation plan for its device drivers.

Binary Transparency. One different plan to help involves an audit log that keeps track of all published binaries.

This helps prevent in particular targeted attacks—for example, if some adversary has a specific target in mind and compromised the distribution of the Linux kernel, they would likely be immediately noticed if they introduced a backdoor into Linux for the whole world. However, if they were able to introduce a backdoor and distribute that backdoored version only to their target, the adversary would be much more likely to evade detection. If clients check their received binary against the publicly available one before installing it, this personalized attacks can be avoided—if the attacker wants to change the binary for someone, they will need to change it for everyone.

4 *Booting the System*

4.1 *Secure Boot*

In order to actually run an application, we rely on large amounts of software running on our computer, from the applications themselves to the operating system that supports them. If the operating system itself is compromised, for example, the modified OS could undermine all of the defenses we just discussed. "Secure Boot" is one strategy for getting some partial protection against these attacks.

Devices using secure boot have a small amount of read-only memory (ROM) that contain a small piece of code that runs on boot. This boot-ROM code has a signature-verification key baked into it. There is no way to change this key—it is a fixed part of the hardware Booting then involves several steps:

1. On boot, the CPU will begin running the hardcoded Boot ROM code, which has a signature-verification key $\mathsf{vk}_{\mathrm{ROM}}$ hardcoded into it.

2. the Boot ROM will load the code for another layer called the *boot loader*. The boot ROM will then verify that the boot loader code carries a correct signature that verifies under $\mathsf{vk}_{\mathrm{ROM}}$. If the signature is valid, the boot ROM code will begin executing the boot loader. The bootloader has another signature-verification key ($\mathsf{vk}_{\mathrm{bootloader}}$)) baked into it.

3. The boot loader will load the code for the operating system and verify it using $\mathsf{vk}_{\mathrm{bootloader}}$. If the signature is valid, the bootloader will redirect control to the operating system.

This way, the system can verify that only boot loaders approved by the hardware manufacturer can run on the machine. These boot loaders then can verify that only trusted operating systems are executed.

Many systems use secure boot: iPhone, Android, chromebooks, game consoles, and UEFI secure boot on PCs.

In some cases (e.g., UEFI secure boot) secure boot is a mechanism to protect against malware that tampers with the operating-system code. While the malware may be able to compromise the running machine, after rebooting the machine, the user has some assurance that it is running an uncompromised operating system.

In other cases (e.g., game consoles) secure boot is a mechanism to prevent the device owner from installing a non-standard operating system on the device. Game-console vendors often sell the console hardware at a loss, and they make their money back by selling game

A number of researchers have used PlayStation 3 consoles for brute-force password-cracking and cryptanalysis (e.g., factoring). Game consoles often have a large number of CPU/GPU cores, which make them appealing hardware for applications that benefit from massive parallelism.

software. They have a strong incentive then to prevent users from buying game consoles and using them for non-game purposes.

4.2 Measured Boot

Secure boot assumes you know which key is needed to sign the software. But what if the hardware cannot determine which software is good and which is bad? The idea is to measure what software is being booted. This involves hashing the bootloader, then hashing the OS kernel, and finally hashing the OS image. This idea is called measured boot.

While measured boot cannot prevent bad software from loading, different software generates different secrets. The system contains a secret that is durably stored in hardware and remains consistent across reboots. When the system boots up, it derives a secret key based on this hardware secret, using a derivation method that incorporates the hashes of the bootloader, and OS kernel, and perhaps other software.

The OS then uses this secret key to decrypt its data and authenticate to remote servers. Booting a different OS —- perhaps due to malware corruption —- will result in a different key being generated, and this is detected when the data cannot be decrypted.

5 Secure attention key

When to approach a terminal and type your bank password into it, how do you know that you are typing the password into the banking app or into some other app (e.g., the flashlight app) on the machine?

A traditional approach to address this problem is a *secure attention key*: there is a special button or combination of buttons that trap into the operating-system kernel code—interrupting whatever application that may be running.

Windows workstations, for example, required users to type the keyboard combination `CTRL-ALT-DEL` to bring up a login prompt. If a user entered this combination while an app was running, the operating system would interrupt the application and open the legitimate login screen.

Pressing the "Home" button on many smartphones has the same effect.