

# Today

## Code signing

- interpreted code
- native code

## Platforms

# Software installation

In days of yore:

- build the code
- setup programs

apt-get



Coherence:

- installer frameworks
- package managers



6 / 27

Once upon a time (or still today in some Unix environments!) the way to install code was to run `make install` — or worse, `make` followed by copying files into place.

Things evolved a bit with \_\_\_\_\_ that could, well, set up a program. These programs typically ran (at least on PCs) in the days before Biba policy application, so they could generally \_\_\_\_\_.

Eventually, this led to the creation of coherent abstractions for software installation. In Windows-land, Microsoft pulled the rug out from underneath InstallShield by creating the Windows Installer framework, which allowed applications to specify what files they needed to install, registry keys the needed to update, etc., without having to resort to arbitrary code execution at install time. Clean abstractions lead to nice outcomes like idempotency, coherent transactions and rollback/uninstallation, etc. (though you *can* still run arbitrary code via [hilarious workarounds](#)).

In the open-source world, the abstraction of a "package" was typically managed by a *package manager*, which would also add the benefit of dependency management and automatic fetching and installation of dependencies (after all, you only want to run freely-available open source software, right?). Packages could contain arbitrary setup scripts to, e.g., set up new users and groups, but over time we're moving towards having package managers handle these things too so that there doesn't need to be any \_\_\_\_\_. Hooray!

# Software installation today

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
$ curl -sSL https://get.rvm.io | bash
$ curl -L https://omnitruck.chef.io/install.sh | sudo bash
```

## What are the risks?

- running with user privilege?
- running with system privilege?

7 / 27

Having learned all of these lessons (reduce arbitrary code execution as much as possible, use abstractions that can be rolled up into reversible transactions, etc.), we've proceeded to throw them all away over the last decade or so.

Installation scripts that run with user privilege can \_\_\_\_\_.  
\_\_\_\_\_. "But I can inspect the script and make sure it doesn't do anything malicious," you say... sure, but do you?

Installation scripts that run with superuser privilege can \_\_\_\_\_.  
\_\_\_\_\_ That sounds pretty scary! However, \_\_\_\_\_  
\_\_\_\_\_... so what's the critical difference?

## Recall: integrity

Where did that high-integrity software come from?

Where do your software updates come from?

Can we check these things *after* download?

8 / 27

You might *think* you're getting Windows updates from update.microsoft.com, but how do you know? We'll see in the Network Security module that it isn't actually all that hard to spoof domain names on many networks (including Memorial's!), so \_\_\_\_\_

when you install a Windows update? What is your \_\_\_\_\_?

Is there anything that we can do to authenticate software updates *after* we download them in order to \_\_\_\_\_?

# Recall

## Message authentication codes

Allow us to verify things... but what's the problem?

## Digital signatures

$$S = D_{K_S} \{h(M)\}$$

$$V = E_{K_P} \{S\} \stackrel{?}{=} h(M')$$

9 / 27

We can use a MAC to validate the integrity of a message, but only if \_\_\_\_\_. That's not a great fit for such an \_\_\_\_\_ use case as software updates, where one vendor may send updates to millions (or billions!) of end users.

How about digital signatures? A vendor can create a signature over a message (e.g., a software package) and ship that along with the software itself.

Then, anyone who has the vendor's public key can verify that the package was actually sent by that vendor. However, \_\_\_\_\_

\_\_\_\_\_? Moreover, \_\_\_\_\_

\_\_\_\_\_? Do they have to expect \_\_\_\_\_ of their code signing system/team?

# Code signing

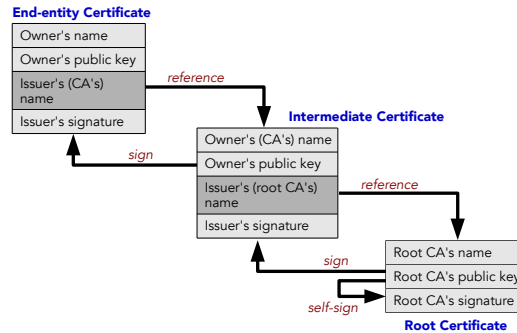
## Chain of Trust

I sign my code with my key  
*Certificate authority* signs that\*  
A *root CA* is ultimately trusted

## Common instantiation: X.509

- standard for representing *who* has signed *what*
- aside: ASN.1 BER originally a telco standard, easy to get wrong

Source: [Wikimedia Commons](#)



10 / 27

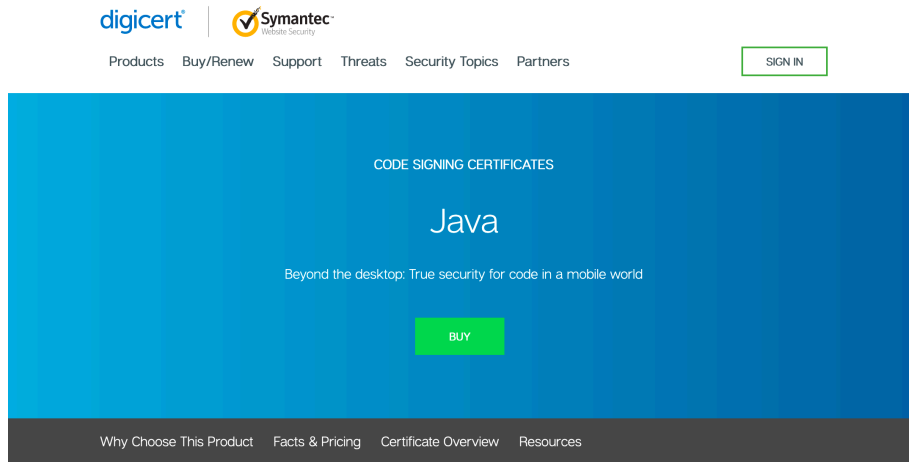
Rather than distributing every vendor's public key reliably, we can distribute a (comparatively) small number of high-value root CA keys. Their corresponding private keys need to be protected, which is part of why we have a chain of trust: \_\_\_\_\_

\_\_\_\_\_. Instead, they \_\_\_\_\_.  
and then \_\_\_\_\_.

ASN.1 is a classic example of an overcomplicated standard make what ought to be simple extraordinarily complex and error-prone. It features *more than one* set of binary encoding rules, and getting those serialization and deserialization rules right has led to a **shocking number of security vulnerabilities**. It shouldn't be this hard, but it is.

# Code signing certificates

## Hype vs fact



11 / 27

Code signing is useful, but like anything to do with security (or computing, or life really), don't believe the hype!

# Purpose of code signing

Goal: verify *identity* of code

“  
*This OS update was released by Microsoft, and I'm choosing to trust Microsoft, so I will choose to trust this OS update*  
”

Note: not verifying *goodness* of code

“  
*This code was signed by Microsoft, so it doesn't have any bugs*  
”

12 / 27

This skeptical attitude towards verification is useful across all of computer security (and, again, across all of life!).

"My bank called and said that..."

Someone who \_\_\_\_\_ called and said that...

"This code is signed, so it must be trustworthy."

This code is signed, so we can attribute its origin to a vendor (or, even better/worse, we can attribute its signature to a root CA who \_\_\_\_\_).

"We can trust this shipping manifest because it's on the blockchain."

This possibly-fraudulent manifest was published before some other stuff was published.  
Probably.

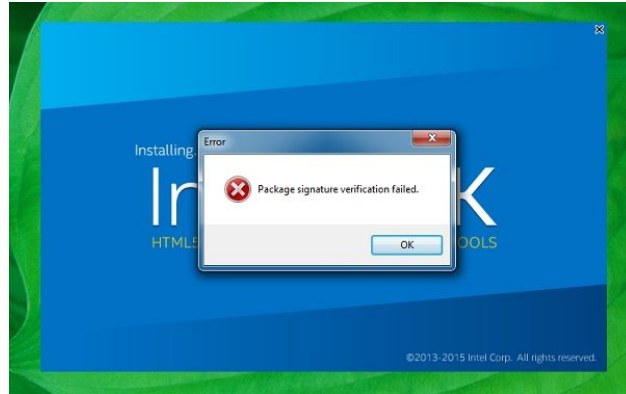


# Code verification

## When?

- installation time
- run time

## How?



13 / 27

We can verify digital signatures on code at install time or at run time. The details of the techniques will differ slightly, but what's really interesting is what \_\_\_\_\_ these two \_\_\_\_\_ provide support for.

# Java

## SecureClassLoader

- loads code like any `ClassLoader`
- adds `CodeSource` and `getPermissions`

Code signing tied to privileges  
(no *confused deputy*)

### getPermissions

```
protected PermissionCollection getPermissions(CodeSource codesource)
```

Returns the permissions for the given `CodeSource` object.

This method is invoked by the `defineClass` method which takes a `CodeSource` as the class being defined.

#### Parameters:

`codesource` - the `codesource`.

#### Returns:

the permissions granted to the `codesource`.

```
CodeSource(URL url, Certificate[] certs)
```

Constructs a `CodeSource` and associates it with

```
CodeSource(URL url, CodeSigner[] signers)
```

Constructs a `CodeSource` and associates it with

14 / 27

Signed JAR files can be loaded and associated with permissions that other code loaded at run time wouldn't have. For example, if I write a Java program that provides run-time plugin support, I can write a `SecureClassLoader` that will give my own plugins (or plugins that I've signed through my online store) permission to access the filesystem, but unsigned plugins may not be able to perform any such operations.

Java provides support for tracking these privileges up and down the call stack, so that the Java runtime can tell whether code is being invoked **on behalf of only privileged code**. This avoids the *confused deputy* problem, in which privileged code is tricked into executing a privileged operation on behalf of malicious and unprivileged code (for example, a malicious plugin calling a legitimate method that saves data into a configuration file, allowing malicious data to be written there).

# Native code signing

That's nice for Java...

... but how about native code?

The processor doesn't verify signatures... who does?

15 / 27

This works really well for bytecode-interpreted languages, where the language runtime is able to interpose itself in the execution of a program.

When executing native code, however, there is no monitor checking every instruction that gets executed. The only thing outside of the binary code which sees the instructions is the processor itself, and it ain't checking digital signatures when you invoke a function!

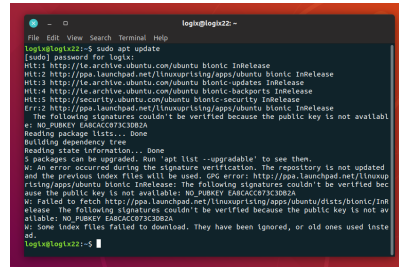
# Signed native code

Installation time: *installer*

Module load time: *kernel loader*

Ordinary applications: *kernel*

Bootstrapping: a bunch of places!



```
login@login22:~$ sudo apt update
[Info] password for login:
Hit:1 http://ft.archive.ubuntu.com/ubuntu bionic InRelease
Hit:2 http://ppa.launchpad.net/linagrip/linagrip/ubuntu bionic InRelease
Hit:3 http://ft.archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:4 http://ft.archive.ubuntu.com/ubuntu bionic-backports InRelease
Hit:5 http://security.ubuntu.com/ubuntu bionic-security InRelease
Err:2 http://ppa.launchpad.net/linagrip/linagrip/ubuntu bionic InRelease
  The following signatures couldn't be verified because the public key is not available:
  #1: NO_PUBKEY E48AC6973C30B2A
Reading package lists... Done
Building dependency tree
Reading state information... Done
5 packages can be upgraded. Run 'apt list --upgradable' to see them.
W: An error occurred during the signature verification. The repository is not updated
and the previous index files will be used. GPG error: http://ppa.launchpad.net/linagrip/linagrip/ubuntu bionic InRelease: The following signatures couldn't be verified because the public key is not available: NO_PUBKEY E48AC6973C30B2A
W: Failed to fetch http://ppa.launchpad.net/linagrip/linagrip/ubuntu bionic InRelease
W: Some index files failed to download. They have been ignored, or old ones used instead.
login@login22:~$
```



Installers can check signatures and require that packages have been signed by someone with a trusted public key. That does, however, lead to funny advice sometimes, e.g.:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
| sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

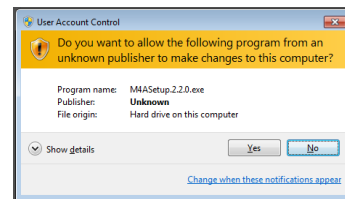
# Windows

Device drivers

Windows update

SmartScreen and EV certificates

Installers and UAC



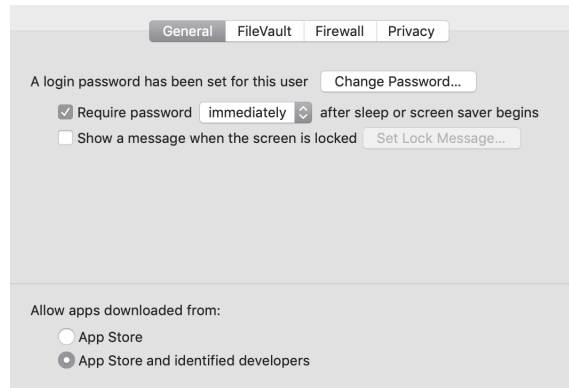
17 / 27

The Windows Quality Hardware Labs (WQHL) initiative didn't so much start as a security initiative as a "stop blaming us" initiative. Hardware vendors aren't always good at software, and buggy device drivers have caused a lot of Blue Screens of Death over the years. Microsoft created the WQHL program to ensure that, if a driver was to be loaded by the Windows kernel, that it would have passed some quality assurance tests first (some of which were actually pretty cool, e.g., proving termination of interrupt handling routines).

# macOS

Not just for drivers!

macOS vs iOS



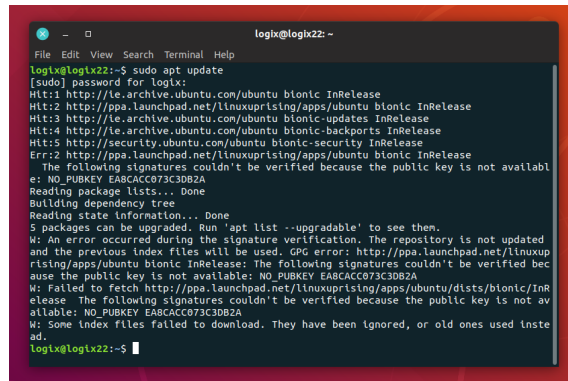
18 / 27

A key distinction between macOS and iOS (at least for now) is that \_\_\_\_\_  
\_\_\_\_\_. The mechanism exists in both, but the policy is  
different. However, \_\_\_\_\_  
\_\_\_\_\_...

# Unix-ey systems

Package managers

Merkle trees



```
logix@logix22: ~  
File Edit View Search Terminal Help  
logix@logix22:~$ sudo apt update  
[sudo] password for logix:  
Hit:1 http://ie.archive.ubuntu.com/ubuntu bionic InRelease  
Hit:2 http://ppa.launchpad.net/linuxuprising/apps/ubuntu bionic InRelease  
Hit:3 http://ie.archive.ubuntu.com/ubuntu bionic-updates InRelease  
Hit:4 http://ie.archive.ubuntu.com/ubuntu bionic-backports InRelease  
Hit:5 http://security.ubuntu.com/ubuntu bionic-security InRelease  
Err:2 http://ppa.launchpad.net/linuxuprising/apps/ubuntu bionic InRelease  
  The following signatures couldn't be verified because the public key is not availabl  
e: NO_PUBKEY E8BCACC073C3DB2A  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
5 packages can be upgraded. Run 'apt list --upgradable' to see them.  
W: An error occurred during the signature verification. The repository is not updated  
and the previous index files will be used. GPG error: http://ppa.launchpad.net/linuxup  
rising/apps/ubuntu bionic InRelease: The following signatures couldn't be verified bec  
ause the public key is not available: NO_PUBKEY E8BCACC073C3DB2A  
W: Failed to fetch http://ppa.launchpad.net/linuxuprising/apps/ubuntu/dists/bionic/InR  
elease The following signatures couldn't be verified because the public key is not av  
ailable: NO_PUBKEY E8BCACC073C3DB2A  
W: Some index files failed to download. They have been ignored, or old ones used inste  
ad.  
logix@logix22:~$
```

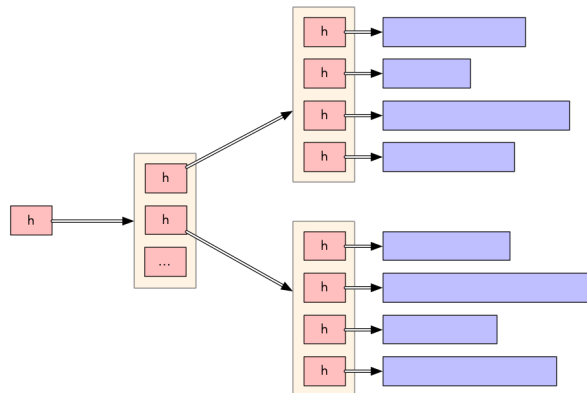
19 / 27

Signing only the root of a package tree is a nice example of a Merkle DAG (which is also a key technology used in other places, e.g., copy-and-write filesystems and blockchains). If you sign a hash of a bunch of hashes of a bunch of hashes, you can effectively sign \_\_\_\_\_ with a single signature!

# Merkle trees

Commitment

Layers



---

Merkle, R. C., "A Digital Signature Based on a Conventional Encryption Function", *Advances in Cryptology — CRYPTO '87*, 1987. DOI: [10.1007/3-540-48184-2\\_32](https://doi.org/10.1007/3-540-48184-2_32).

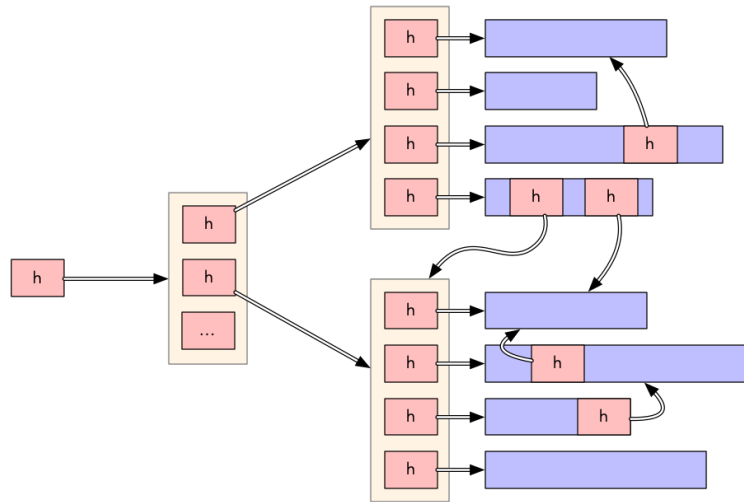
21 / 27

A cryptographic hash can be used to express a *commitment*: without revealing any data now, I can promise you what data I will reveal in the future. When I reveal the data, you can check that it matches the commitment using the hash that I already gave you. If I change even a single bit in the data, it will alter the hash in dramatic ways.

A hash of data allows us to commit to that data. A hash of a bunch of hashes of data allows us to commit to all of the data. We can add more layers arbitrarily to this tree, allowing a single hash to speak for arbitrary amounts of data.



# Merkle DAGs



22 / 27

Technically, a Merkle tree can actually be a more general DAG. This is because various pieces of data within the DAG can actually reference each other, but never with cycles: the only way to generate a hash  $h(X)$  is to have  $X$  in its entirety, so  $X$  can't include a hash that refers back to  $h(h(X))$ .

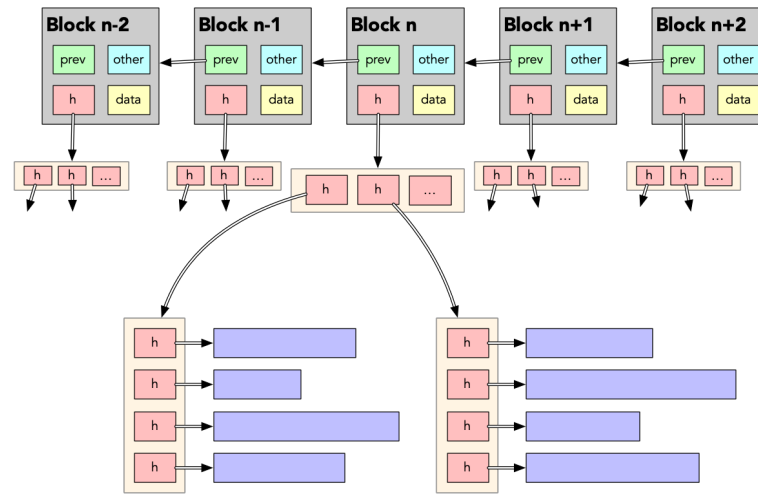
## Aside: blockchain

Hashes

Merkle trees

Blocks

Permissions



23 / 27

Hash functions are now old hat to us. Nothing new here.

Using Merkle trees, each block in the blockchain can refer to an arbitrary amount of data. This could represent shipping manifests, quasi-financial transactions or anything else you might care to think of.

The new thing about a blockchain is that it has \_\_\_\_\_ which represent a \_\_\_\_\_ of data added to the chain. This \_\_\_\_\_ doesn't depend on timestamps per se, it depends on the logical \_\_\_\_\_ relationship that's indicated by a hash function: in order to hash the previous block, it must have been available to you \_\_\_\_\_ (which went into the new block).

The tricky thing about anything "canonical" is deciding \_\_\_\_\_. People can get **surprisingly detailed about these questions when discussing things like comic books**. If we're going to establish a canonical serialization of all the transactions that have occurred in our new crypto-nerd utopia, who gets to decide whether my preferred transaction ordering is correct or yours is?

In a *permissioned* blockchain, we can express authority to say "this happened next" via cryptographic mechanisms like digital signatures. Every client can check, "was this signed using a public key whose certificate was signed by an appropriate authority?" In an *unpermissioned* blockchain, we need some other way to determine who gets to say what comes next. Public blockchains like Bitcoin and Ethereum (at least for now) use a \_\_\_\_\_

scheme in which whoever can solve a cryptographic puzzle: find  $x$  such that  $h(prev, x)$  starts with at least  $n$  zeroes. As  $n$  increases, this requires an obscene amount of energy for computation, so much so that some people want to use all of Muskrat falls to make a small dent in the global demand. Ethereum might ever switch to a \_\_\_\_\_ scheme in which all of the most well-moneyed interests get to say what's what... and that's better than fiat currency how? (/end skeptical rant)

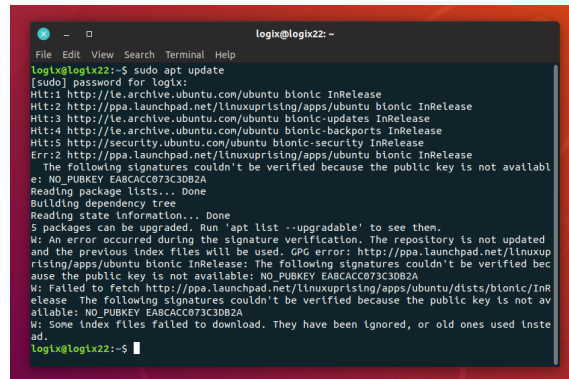
# Unix-ey systems

Package managers

Merkle trees

**verifexec**

... but where does it all start?



```
logix@logix22: ~  
File Edit View Search Terminal Help  
logix@logix22:~$ sudo apt update  
[sudo] password for logix:  
Hit:1 http://le.archive.ubuntu.com/ubuntu bionic InRelease  
Hit:2 http://ppa.launchpad.net/linuxuprising/apps/ubuntu bionic InRelease  
Hit:3 http://le.archive.ubuntu.com/ubuntu bionic-updates InRelease  
Hit:4 http://le.archive.ubuntu.com/ubuntu bionic-backports InRelease  
Hit:5 http://security.ubuntu.com/ubuntu bionic-security InRelease  
Err:2 http://ppa.launchpad.net/linuxuprising/apps/ubuntu bionic InRelease  
The following signatures couldn't be verified because the public key is not availabl  
e: NO_PUBKEY E8BCACC073C3DB2A  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
5 packages can be upgraded. Run 'apt list --upgradable' to see them.  
W: An error occurred during the signature verification. The repository is not updated  
and the previous index files will be used. GPG error: http://ppa.launchpad.net/linuxup  
rising/apps/ubuntu bionic InRelease: The following signatures couldn't be verified bec  
ause the public key is not available: NO_PUBKEY E8BCACC073C3DB2A  
W: Failed to fetch http://ppa.launchpad.net/linuxuprising/apps/ubuntu/dists/bionic/InR  
elease The following signatures couldn't be verified because the public key is not av  
ailable: NO_PUBKEY E8BCACC073C3DB2A  
W: Some index files failed to download. They have been ignored, or old ones used inste  
ad.  
logix@logix22:~$
```

Once you've installed your signed software, it's also possible to check digital signature at run time when you execute a program. JunOS / NetBSD / FreeBSD have a **verifexec(1)** scheme that allows execution to be limited to signed files only. If you're running an embedded appliance running a large portion of the Internet backbone, you probably want to ensure that only your code is running on that appliance! However, much like Data Execution Prevention (DEP, used for `W^X` and `noexecstack`), that doesn't prevent an attacker from "living off the land" if they manage to subvert your code. It does, however, make their job harder, which is a worthwhile thing.

# Boot process

(a.k.a., *bootstrapping*)

"Secure" boot, "verified" boot, "measured" boot...

*Trusted Computing* initiative

UEFI, "Certified for Windows 10/RT"...

25 / 27

The term "secure" boot is, unfortunately, a bit ambiguous. It can be used to mean one of two \_\_\_\_\_ things.

*Verified boot* means that a hardware component called the \_\_\_\_\_ (TPM) gets involved in the boot process. This allows the bootloader's digital signature to be checked and policies such as "you must use a bootloader signed by Microsoft" can be \_\_\_\_\_. If that initial "root of trust" verification fails, the system doesn't boot.

An alternative is called *measured boot*, in which signatures are checked and the results are \_\_\_\_\_ for later inspection by software. Nothing stops the computer from booting with unsigned code, or code with an incorrect signature, but software can later check to see what code booted it. In particular, the TPM can provide its "measurement list" as part of a \_\_\_\_\_ procedure, allowing, e.g., a server to only accept connections from computers booted from specific software signed by specific vendors.

This is all part of the *trusted computing* initiative, which was enormously controversial when it was introduced. On the one hand, "trusted computing" could be help identify what code was running where, which could have some security benefit. On the other hand, it could also be used to prevent users from accessing DRM-protected content in "unapproved" ways or even prevent users from running "unapproved" OSes. That is to say, the ownership model of your computer would look more like that of your (non-jailbroken) phone: it wouldn't really be \_\_\_\_\_.

These days **Microsoft has a program** via which they will sign an open-source bootloader like GRUB for a small fee. This allows even systems locked down with mandatory verified boot to run non-Windows operating systems \_\_\_\_\_.

# Summary

## Code signing

- interpreted code
- native code

## Platforms