Boots as builds and boot security — a DeBoot report

Andrew Macpherson and Aata Hokoridani

December 6, 2023

Abstract

We consider bootstrapping computers, also known as bare metal provisioning, in a more general context of deployments and dependent builds. We review the state of the art, exhibit a functional programming inspired data model that expresses builds declaratively and specialize it to a boot context, and discuss the need for and components of a trust-oriented security model for provisioning.

1 Introduction

Booting computers has long been a somewhat neglected corner of IT infrastructure. It is often easy to ignore the sequence of unfamiliar and usually poorly documented ephemeral stages a computer quickly steps through between a power cycle and normal operation. Indeed, one usually expects that a correctly configured system spends only a tiny fraction of its life cycling through these stages.

Despite this, there are a few reasons one may wish to expend attention on the bootloader. Paramount among these is *security.* In order to evaluate the security of the software supply chain deployed in a computer system, one must trace the root of trust all the way to the beginning of a power cycle. This entails careful formulation and analysis of the security invariants that ought to be preserved during the execution and handoff of each stage of the boot sequence.

A second, related issue is *determinism*, that is, the idea that the same boot process configuration will yield the same system state transition regardless of context. To this end, both compilation and building of the deployed software and execution of initialization code must be reproducible from a clear task description that makes all inputs explicit. Reproducibility of execution can be detected in practice by making careful *measurements* of system state at critical stages.

We think the bootloader is overdue a modern theoretical treatment. In this article we address the question:

> What type of thing is a boot sequence?

An answer to this question empowers us to draw analogies with other types of computation and make more precise and interpretable security claims. A well-defined data model for defining boot sequences can have a number of benefits:

- *Accessibility.* A modern, implementation-agnostic description language means more users understanding and taking control of their own boot sequences.
- *Security.* A serialized boot definition can be used as a part of a *software bill of materials* which is the subject of security claims about a deployment.
- *Extensibility.* The present obscure state of documentation makes it difficult for developers to integrate new technologies such as novel storage backends early in the boot process.

The second question we address in this article is:

> How can boots be trusted?

The answers to these questions cannot be divorced from one another. In order to make trustworthy claims about boots, we have to be able to clearly state the subject of the claims. On the other hand, a formal definition of a boot cannot have much practical significance if users cannot be sure that the state reached by their system matches the definition.

In §2 we introduce a data model for defining computations and exhibit how typical boot sequence paradigms can be expressed in this model. In §3 we discuss elements that would be needed for a formal cybersecurity framework and review some ubiquitous examples with this in mind.

1.1 Related work

For a thorough engineering overview of the structure of boot processes at the firmware level, see [15].

There are many cross-industry initiatives to develop standards associated to the boot process:

- The *Universal Extensible Firmware Interface* (UEFI) is a set of cross-platform standards for an execution environment presented by platform firmware which early stage boot manager or bootloader software can hook into.[1]
- The *Preboot Execution Environment* (PXE, pronounced 'pixie') is a component of the UEFI specification that defines a protocol, based on DHCP and TFTP, for negotiating a network boot. It is the standard way to load the next stage after platform firmware over a network, although some systems also offer boot from a webserver.
- The *Extended Bootloader Base Requirements* (EBBR) are a subset of UEFI designed as a specification compatible with U-Boot, the industry standard firmware and bootloader implementation for embedded systems.[2]
- The *multiboot* specification and its sequel does some work to define the format of OS kernel loaders. Unlike the preceding examples, multiboot does not attempt to specify how systems should operate at the platform firmware level — its scope is limited to starting an OS. The reference implementation of a multiboot-compliant host is the GRUB boot manager, widely used in booting Linux or BSD on x86 computers (although Linux and BSD are not usually distributed as a multiboot-compliant client!)
- The *boot loader specification* project defines disk formats, file formats, and naming conventions for boot loader menu entries.[3] BLS-compliant menu entry configurations are intended to be OS-agnostic and can be thought of as a data model for a kernel loader boot stage. However, boots defined in this format are limited to ones where the boot artifacts are located on local storage, which must be partitioned in one of a few pre-defined layouts. Thus *extensibility* cannot be said to be one of the goals of the project.

Applications designed for bare-metal provisioning of fleets of devices in data centres must define their own data model for specifying deployments. Generally speaking, these systems operate over a network (hence use network-based boot protocols, the major one being PXE) and are opinionated about the structure of boot flow. Some examples are OpenStack's *Ironic*,[4], Canonical's *MaaS*,[5], and Equinix's *Tinkerbell*.[6]

The OpenStack authors have written on the need for a standardized bare metal provisioning API [1].

The early boot and system initialization standards are accompanied by standards defining data models and serialization formats for hardware devices. These models aggregate the addressing systems of various bus types (e.g. PCI) into a single namespace. As well as providing essential input data for boot loaders to discover and initialize peripherals, a path-like serialization format is useful for specifying the location of boot images and configuration in boot definitions.

- The UEFI specification includes a systematic and quite general language of *devpaths* for defining *locations* for sourcing images.[7]

  A related standard is the *Advanced Configuration and Power Interface* (ACPI) device model. ACPI is also maintained by the UEFI Forum and can be invoked within UEFI devpaths, but is more commonly used for initialising a booted operating system.

[1] https://uefi.org/specifications
[2] https://arm-software.github.io/ebbr/
[3] https://uapi-group.org/specifications/specs/boot_loader_specification/
[4] https://wiki.openstack.org/wiki/Ironic
[5] https://maas.io
[6] https://tinkerbell.org/
[7] https://uefi.org/specs/UEFI/2.10/10_Protocols_Device_Path_Protocol.html

- The Devicetree specification provides an alternative data model and accompanying source and binary formats for informing bootloaders about hardware layout.[8]

Regarding reproducibility, the subject of *reproducible builds* in software distribution has already received a fair amount of attention [12].[9] As for reproducibility of system state, the approach of *measurements* in boot sequences is well established and widely implemented in standard hardware [11].[10]

2 Boots as builds

A boot can be regarded as a set of steps to compute the state of an initialized system. In some cases, recomputing the value with updated inputs (i.e. rebooting) can be accelerated using a caching strategy, so that a reboot may occur without necessarily going through a full power cycle. In some cases, the requirements to launch a boot stage can be provided by more than one environment, so that the same target can be reached in multiple ways.

In §2.1, we define a data model rich enough to specify describing builds with multiple valid routes to the target. Subsequently, we show several ways to express some common boot pathways in this model.

2.1 Data model

2.1 Definition. An *n-ary operation* from types $(T_1, \ldots, T_n)$ to type T_0 is defined to be a function $f : \prod_{i=1}^{n} T_i \to T_0$.[11] Note that the (single) output and inputs of an n-ary operation are indexed together, in that order, by $[n] := \{0, \ldots, n\}$.

A *task* is an operation f of any arity n together with a labelling $\texttt{name} : [n] \to \texttt{String}$ of the inputs and output by strings and a "task description," which should be an explicit expression of some n-ary operation comprehensible in the context in which the task will be used. The label $\texttt{name}(0)$ of the output of the operation is called the *target* of the task.

A *build specification* or simply *build* is a set of tasks.

Expressed in a Python-like pseudocode, the data schema looks something like the following:

```
[
  (name_0, f, [name_1, ..., name_n]),
  ...
]
```

In this picture, the task description would be a Python function and the labelled source and destination types are denoted by (typed) named variables.

We express the same data more verbosely in a JSON format:

```
[
  {
    "target": {"label": "name_0", "type": "T_0"},
    "inputs": [
      {"label": "name_1", "type": "T_1"},
      ...,
      {"label": "name_n", "type": "T_n"},
    ],
    "task": @td
  },
  ...
]
```

[8] https://www.devicetree.org/
[9] https://reproducible-builds.org
[10] https://trustedcomputinggroup.org/
[11] For the purposes of this paper, a 'type' can be understood as a set.

Here `@td` should be some task description schema whose details we leave unspecified.

2.2 Definition. Let B be a build and t a target of B, and $(i_1, \ldots, i_k)$ a set of labels. A *solution* of $(B, t, (i_1, \ldots, i_k))$ is a subset S of B such that:

- t is a *root* of S, that is, it is the unique target contained in S that is not also an input.
- $i_1, \ldots, i_k$ are the leaves of S, i.e. they are the only inputs that are not also targets.
- Every target in S is the output of at most one rule in S.

In other words, a solution of $(B, t, (i_1, \ldots, i_k))$ defines a way (but not necessarily a sequential order) to compute an element of T_t given elements of $T_{i_1}, \ldots, T_{i_k}$.

2.3 Example (Makefiles as build specifications). We can express the data of a build specification in a Makefile-like syntax as follows:

```
1 name_0: name_1, ... name_n
2     echo $(f name_1 ... name_n) > name_0
```

This can be understood as a genuine Makefile if:

- All the inputs and outputs are "files;" that is, $T_i = \texttt{File}$ for all i. For the purposes of Make, a *file* can be understood as a tuple $(\texttt{name}, \texttt{mtime}, \texttt{data})$.
- `f` is a shell function.
- Each label appears as the output of at most one step. That is, no step has multiple 'providers'.

Recipes in real-life Makefiles (and indeed any build system) are often not pure functions, but rather depend implicitly on some system state whose scope varies on a per-target basis. If one's goal is determinism, all quantities on which the computation actually depends must be listed as explicit inputs.

2.4 Example (Dependencies). A specialization of the build specification idea is that of *dependencies* used in package management. When installing a package, no explicit function is used to compute the installation of a package from its dependencies; the dependencies simply need to be installed on the system for the package to function. Such systems are not rich enough to express a package that 'depends on' another in multiple ways. Hence, in a dependency specification one expects that there be at most a single task connecting each tuple of input labels to each output label. However, a label may still be the target of more than one task.

2.5 Example (DAGs and computation). A build specification does not encode a single computation, but rather potentially multiple ways that a particular function can be computed. A solution of a build specification is a particular *declaratively specified) computation realising the build.

Multiple providers Unlike DAGs, our data model is flexible enough to describe builds in which a label can be the target of multiple tasks, each considered as good as any other. In the language of dependency management, our dependency formulae are allowed to contain logical disjunction as well as conjunction.

Disjunctive dependencies are extremely common in practical deployment scenarios such as RPM or DEB-based package management, where objects or configuration fragments listed as a dependency of a given package may have several provider packages, only one of which need be installed to satisfy the dependency.

Naturally, examples of multiple providers in the boot process are also plentiful:

- The feature `EFI_BOOT_SERVICES` is provided by any UEFI-compliant firmware (for example Tianocore EDK2), but also by any firmware complying with the more stripped-down EBBR specification, such as U-Boot.
- HTTP and other network drivers are available in UEFI, GRUB, or a Linux initramfs.
- Common Linux filesystems such as Ext4 and BTRFS each have providers (drivers) in the form of GRUB and Linux modules, but are not available in UEFI.
- An IP address can be derived either from a static configuration option or via DHCP, either of which itself has several providers spread across different boot stages.

Multi-stage builds A boot system requires multiple computations that take place at different times and in different contexts. These stages can be expressed as partial evaluations of a single computation that takes, for example, the development environment, the target device, and the user requirements as input. The targets of the associated build specification may be partitioned into connected components labelled by the build stage during which they are evaluated.

2.6 Example (Caching state to persistent storage). Suppose we have a division of a computation into "build time" and "run time." The system must undergo a power cycle between these stages, so they cannot share any memory state. Suppose some quantity `blob` available at build time needs to be made available at run time; it must be cached to persistent storage readable by the run time computer.

To model this, we posit an $\mathbb{N}$-indexed variable `storage_s[]` that tracks the state of all attached persistent storage devices. We can consider it as a list of tuples *(device,state)*, where *state* comprises some metadata and a large byte array. The build environment provides a driver which yields a handle `dev_h` to an attached storage device — essentially a method table with `load` and `save` implementations. These can be thought of as returning *commands* to the physical device.

A state update to storage lists as follows:

```
[
  # build time
  (tx, save, [dev_h, blob, path]),
  (storage_s[1], commit, [storage_s[0], [tx]])
]
```

We have explicitly noted pre-build `storage_s[0]` and post-build `storage_s[1]` storage device state.

Then, at runtime we can retrieve it.

```
[
  # run time
  (q, load, [dev_h_, path]),
  (blob_, query, [storage_s[1], [q]])
]
```

Note that at run time we have a different instantiation of the device handle `dev_h_`, and possibly even a different *type* of handle if the run and build environments have different driver models. Moreover, although we certainly hope that the retrieved value `blob_` equals the saved value `blob`, this property is not guaranteed syntactically — see §3.

2.2 Boot sequences as build specifications

In the following examples, we use the Pythonic notation and explain the typing, for which there are often multiple reasonable approaches, and task descriptions in prose.

How can examples like these be applied in practice? The procedure followed in the examples is:

- Mechanistic in nature, and can easily be followed manually by a human programmer with access to the necessary information or even automated;
- Provides a clear specification language for boot sequence implementors.

2.2.1 Loading an EFI application

Suppose we want to boot a system into an EFI application stored as an EFI executable on some removable medium.[12] From a fresh system to a full initialization proceeds in several stages:

1. *Install firmware.* Build and install firmware to NVRAM.
2. *Install payload.* Build and install EFI application to bootable media.

[12] Cf. https://wiki.osdev.org/UEFI#UEFI_applications_in_detail

3. *Configure.* Write boot options including target device, partition number, and filesystem path to NVRAM.
4. *Boot.* Power on the device and, where necessary, capture user input.

Specification of these tasks usually fall to different entities:

1. The OEM usually ships devices with firmware pre-installed.
2. The user or customer-facing vendor builds and installs the payload.
3. The user configures the UEFI boot menu.
4. Firmware developers define the initialization process and handoff.

However, in principle there is no reason for this task allocation other than some tasks (such as firmware installation) are generally considered too difficult for ordinary users to pull off.

To describe the process in more detail, we suppose further that the removable drive is a USB drive formatted with a GUID partition table,[13] and the EFI application is stored as an EFI executable file on a VFAT-formatted EFI system partition on the drive. Our target is the computation of a full system state `init`.

An abstracted, implementation-agnostic form of the boot process might look as follows:

```
# EFI application boot, abstract
[
  (init, efi.exec, [payload]),
  (payload, vfat.read, [boot_fs, boot_path]),
  (boot_fs, vfat.mount, [boot_part]),
  (boot_part, gpt.get_part, [boot_dev, boot_partnum]),
  (boot_dev, usb.get, [boot_devpath])
]
```

In prose, the system needs to be able to:

1. Locate, initialize, and obtain a read interface `boot_dev` to the USB device.
2. Parse a GUID partition table and locate a partition `boot_part` selected by partition number.
3. Mount and read from a VFAT filesystem `boot_fs` stored on the partition.
4. Execute an EFI application `payload` saved on `boot_fs` in EFI executable format.

The system also needs to be informed values of the leaves of this build specification, that is, the values of `boot_path`, `boot_partnum`, and `boot_devpath`. More precisely, these values need to be recorded somewhere accessible to the early boot environment and in a format understood by the same.

Acknowledging that the values of `efi.exec` and the other functions in the preceding listing are merely abstract interfaces for which we must provide an implementation (i.e. a driver), we may rewrite the above as follows:

```
# EFI application boot, concrete
[
  (init, apply, [efi.exec, payload]),
  (payload, apply, [vfat.read, boot_fs, boot_path]),
  (boot_fs, apply, [vfat.mount, boot_part]),
  (boot_part, apply, [gpt.get_part, boot_dev, boot_partnum]),
  (boot_dev, apply, [usb.get, boot_devpath])
```

All of the listed driver functions are now leaves for which we must specify *providers*. Typically, the individual functions will be made available, along with other related functions, as an entry in a function table provided by an associated driver.

[13] https://uefi.org/specs/UEFI/2.10/05_GUID_Partition_Table_Format.html

```
    (efi.exec, get, [Impl(EFI_BOOT_SERVICES)]),
    (vfat.read, get, [Impl(VFAT)]),
    (vfat.mount, get, [Impl(VFAT)]),
    (gpt.get_part, get, [Impl(GPT)]),
    (usb.get, get, [Impl(USB)])
```

Merging further, multiple standards guarantee implementations of all of the required drivers:

```
    (Impl(EFI_BOOT_SERVICES), get, [Impl(UEFI)]),
    (Impl(EFI_BOOT_SERVICES), get, [Impl(EBBR)]),
    ...
```

This yields our first instance of multiple providers of the same interface. We can continue the same idea by listing multiple implementations:

```
    (Impl(EBBR), get, [U-Boot]),
    (Impl(UEFI), get, [EDK2]),
    ...
```

Returning to the leaf data, we usually need to obtain the device from some commonly understood string encoding such as a Devicetree path, or possibly bundled together as in a UEFI device path.The UEFI specification asks that the boot manager read such paths from a table of options *Boot0000*, *Boot0001*, and so on, in an order defined by the variable *BootOrder*.[14]

```
    (boot_path, efi.devpath.get_fs_path, [boot_efi_devpath]),
    (boot_partnum, efi.devpath.get_partnum, [boot_efi_devpath]),
    (boot_dev, efi.devpath.get_dev, [boot_efi_devpath]),
    (boot_efi_devpath, select, [fw.efivars.boot, bootnum])
    (bootnum, select, [fw.efivars.bootorder, 0])
]
```

The new leaves are the EFI variables mentioned above which the user may configure in a firmware configuration utility. A modern boot manager should be able to facilitate this process by scanning available devices and populating the *Boot* array automatically, so that the user need only configure *BootOrder*.

2.2.2 Loading Linux with a kernel command line

Suppose now that we wish to load and jump into a Linux kernel from the GRUB boot manager. A typical invocation of a Linux kernel makes use of the following inputs:

1. To initialize the state, we need a Linux kernel (a binary blob) and a kernel commandline (a string). The kernel command line may have many arguments, but at the very least it must have a `root=` argument, which may be one of a few permitted forms.
2. If an initramfs is used, as it is for all but the most elementary kernel invocations, we need the initramfs binary blob as well.
3. In some cases, the loader may also require a Devicetree blob describing the layout of hardware.

We can express this boot naïvely as follows:

```
# Initramfs boot with requirements deduced from kernel commandline
[
  (init, apply, [load, kernel, initramfs, root, args]),
  (load, get, [Impl(Loader)]),
```

[14] https://uefi.org/specs/UEFI/2.10/03_Boot_Manager.html

In practice, the two or three binary blobs needed to boot are written in advance — call it "install time" — to a persistent storage medium that the loader can read, as in Example 2.6. The format in which they are written depends on the loader, for example:

- (GRUB). The kernel and initramfs are made available to the loader by storing them both on some filesystem accessible thereto. In GRUB, this would be the value of the `$root` variable.
- (U-Boot). The kernel, initramfs, and device tree blob are bundled together along with configuration into a unified binary "FIT" image.[15]

The path or paths to these data are then written, along with the textual metadata (i.e. kernel command line), into bootloader configuration.

Let us focus our attention now on the case of GRUB, and try to imagine how we might automate the process of identifying a suitable peripheral to use as the GRUB root. A suitable peripheral consists of:

- A physical attached device. For the purposes of this example, we assume the device is an attached mass storage medium rather than an abstraction such as a network protocol.
- A GRUB driver for the attached device, either compiled into GRUB or in the form of a dynamically loaded GRUB module.
- A driver for the attached device available in the build environment, so that the data can be written there at build time.

The build system must write the kernel and initramfs to such a device and incorporate the associated `grub_root` and filesystem paths `kernel_path` and `initramfs_path` into GRUB's configuration. These values can go either in the main configuration file *grub.cfg* or into separate BLS configuration fragments.[16] The location of the BLS fragments must then be built into *grub.cfg*.[17]

Now we can expand the computation executed by GRUB, observing how it provides the leaves `kernel`, `initramfs`, `root`, `args`, of the fragment listed above and exposes its own leaves `bls_path`, `grub_root_path` which must be provided by configuration, and storage state `storage_s[1]` (cf. Example 2.6) which must be prepared by installation.

```
    # GRUB runtime
    (Impl(Loader), get, GRUB),
    (kernel, load_kernel, [grub_root_h, storage_s[1], kernel_path]),
    (initramfs, load_initrd,
        [grub_root_h, storage_s[1], initramfs_path]),
    (root, bls.root, [bls]),
    (args, bls.args, [bls]),
    (kernel_path, bls.kernel_path, [bls]),
    (initramfs_path, bls.initramfs_path, [bls]),
    (bls, load_bls, [grub_root_h, storage_s[1], bls_path]),
    (grub_root_h, load_h, [grub_root_path, grub_root_driver]),
]
```

We could go further and add new stages "kernel build time" and "initramfs build time," expanding the compilation of the kernel and initramfs by deducing driver requirements imposed by the `root=` kernel argument.

2.2.3 Multi stage boot

In all but the simplest scenarios, boots need to go through several stages because the resources needed to fetch the operating system need codecs or drivers not available at the first stage. To illustrate the multiplicity of possibilities, we enumerate here some of the more common elements that may be used to construct a boot.

[15] https://docs.u-boot.org/en/latest/usage/fit/source_file_format.html

[16] https://uapi-group.org/specifications/specs/boot_loader_specification/#type-1-boot-loader-specification-entries

[17] These configuration files do not necessarily reside on the same device as `grub_root` — indeed, in x86 Linux systems it is common for them to be located on a different partition.

1. Booting a program that resides on a filesystem other than FAT; for example, `ext4` or `btrfs`. The target `btrfs` is provided by either GRUB or Linux, but is not part of the UEFI spec.
2. Booting an ELF executable and not an EFI PE; provided by GRUB or Linux.
3. Boot a kernel that lives on an attached storage device whose filesystem may be any of a list of supported filesystems. The actual format is autodetected on startup (a dynamic dependency).
4. Boot a Linux image fetched from a novel storage backend such as Swarm, IPFS, or Arweave. Software for the storage backend is available for Linux. In the context of §2.2.2, the client software can be listed as a "driver" for the initramfs build.
5. Boot that waits to capture user input interactively (a dynamic dependency). Various models allow the user different levels of expressivity:
   - Selection from several hardcoded boot options — provided by UEFI.
   - Selection from several boot options with an interactive editor to modify them. Dependency is `grub`.
   - Shell with various capabilities. Available in UEFI, GRUB, U-Boot, Linux initramfs...

Network boots open up many more dimensions of configurability.

2.7 Example (HTTP boot). Fetch an image from the WAN over HTTP with static network configuration. Requires network interface drivers, a TCP/IP stack, and HTTP.

1. Same as above but with HTTPS. In addition to the above you need TLS and root certificates (as trust anchor or with trust derivable from a trust anchor). Typically this would be used with DNS.
2. HTTP boot but also verifying a PGP signature. Requires a PGP implementation and the signing pubkey (as trust anchor or with trust derivable from a trust anchor).
3. HTTP boot with authentication by an alternative signature scheme, such as BLS with signature aggregation. The dependency is the verification libraries for the scheme, which may be available only at the OS level.
4. HTTP boot with network layer autoconfiguration — a dynamic dependency. Provided by DHCP, DHCPv6, or RA.
5. HTTP boot, but the image URL is a DNS name rather than an IP address. Requires a DNS resolver. If using DHCP, requiers a DHCP client plugin that registers the DNS server address field.

3 Security

How can we trust boots? To approach the question, we must first make it more precise:

> How can we *estimate* and *optimize* the probability that, given a set of assumptions about the environment and the capabilities of an adversary, booting a system puts it into a state with desirable properties?

The 'desirable properties' of the booted system state are our *security objectives*; the capabilities of an adversary are the *threat model*.

A practical boot security framework would provide:

- A boot risk management language. A nomenclature for common boot security objectives, assumptions, and threats.
- Methods for evaluating risk.
- Methods for constrained optimization of risk.

For our nomenclature we should draw on the language of firmware and boot loader development practitioners [15] as well as models associated to the various information security frameworks in use across the industry [14].

3.1 Remark. The information security literature abounds with references to mnemonics such as the famous 'CIA triad' of information security *confidentiality, availability, integrity* or Microsoft's STRIDE threat model. While these mnemonics can be useful for generating rules of thumb, they are generally not expressive enough to accurately model interesting security goals in a given context.

3.1 Building a risk model

In this section, we review standard approaches for formulating *claims* made about software and linking them to trusted *authorities*. We argue that these collectively form ways to build an 'outsourced' risk model for initializing a computer system in a given context. It is not always possible to verify the content of claims *in situ*, but with proper infrastructure one can at least verify that the claim has been made, not repudiated, and bound to an authority with known attributes. Given a set of desired properties, risk can then be controlled in a running system by validating claims and disabling specific functionality — for example, refusing to boot — in the event of validation failure.

Outsourcing trust Some types of security objectives can be efficiently verified by client systems on-the-fly. For example, the *integrity* of a retrieved boot image can be verified at boot time to high confidence against a known hash.

Other objectives are either infeasible to verify locally — for example, 'an adversary with the ability to run unprivileged software cannot achieve a privilege escalation' — or are informal in nature and cannot in any sense be verified or falsified. When we cannot verify, we need to be able to *trust* boot inputs and processes. This trust is most often inherited from trust in an individual, organization, or brand name — for example, the authors of a software distribution — that stands accountable for the good properties of the artefacts. The confidence we can derive from these claims then becomes a function of the nature of the claim and of our trust in the authority's reliability in making such claims.

Hence, to outsource our risk model to a set of claims made by other trusted entities, we would also need an 'identity system' comprising:

- A scheme for binding identifying attributes, such as common names, to entities, such as natural persons or organisations, that are the subject of trust.
- A certification system that binds credentials, such as a public key, with identifying attributes.
- A set of authentication methods and accompanying credentials for verifying this binding.

Certification and authentication systems can be online or offline. If offline, certificates may ship with an expiry date, and the system may also provide for a highly available *revocation database* that security-conscious systems should query as part of the authentication process.

Examples of such systems:

- Probably the most widespread system in use is the Internet public key infrastructure (PKI) based on X.509 certificates [5]. The process for binding entities to their identities is described by a CA's *Certification Practice Statement* (CPS). PKI and its relation to other classical certification methods such as PGP's web of trust (WoT) model are discussed in [8]. PKI provides for revocation either through certificate revocation lists (CRLs) as introduced in RFC-5280, the newer Online Certificate Status Protocol (OCSP), or browser-specific solutions [13].
- JSON web tokens [10] provides an authenticated scheme to bind attributes, called *claims* in this context, to keys (certification). IANA maintains a list of recognized claim schemata.[18]
- Decentralized identifiers (DIDs) provide a way to communicate and validate authentication material.[19] The handling of certification and name binding tasks is left up to the individual DID methods.

In recent years there is also substantial interest in using the highly available timestamping properties of blockchains to provide name-binding or PKI services [9, 6].

[18] https://www.iana.org/assignments/jwt/jwt.xhtml
[19] https://www.w3.org/TR/did-core/

Claims about artefacts Given an adequate system for identifying the creators of claims, we need a common language and taxonomy for the content of the claims. We recommend adoption of the language of the SCITT working group [3].[20]

Here are some examples of claims about generic software distributions:

- Claim that a software distribution has been published under a certain product name by an owning entity.
- Existence or non-existence of registered vulnerabilities, for example in the form of CVE records.[21]
- Existence and claims of audits bound to a specific software version. This idea has been floated several times in the context of smart contract deployments.
- Claim that a program or library implements a certain interface or fulfils a certain contract.
- Claims of concrete security guarantees in a given context and threat model.

Further examples can be found in the use cases report of the SCITT WG [4].

More relevant to the present paper, we may wish to claim that an operating system distribution image is bootable via a stated boot process, or a boot process with certain properties. For instance:

- Statement of format of the image (raw partition, raw GPT-formatted partitioned disk with EFI system partition) and entry point to the boot process (EFI system partition, explicit start vector)
- List of components used in the boot process.
- Boot computation specification in the sense of §2.
- Contract implemented between bootloader and payload.
- Preservation of privilege level between boot stages, in cases where these are distinguished.
- Boot loader makes use of secure hardware services such as TPM.

3.2 Example: integrity/tampering boot security problem

The most widely discussed security goal of a boot sequence is the *integrity* of boot images as measured against a commitment made by a trusted authority or the operator himself.

3.2 Example (U-Boot standard boot). *Context.* An ARM SoC with (trusted) U-Boot firmware is booting a FIT image,[22] comprising a kernel, initramfs, device tree blob, and kernel commandline. The image is written to an SD card by the operator (cf. Example 2.6).

Adversary. May modify the contents of the SD card after the user writes it but before booting.

Security goal. Either the specified FIT image is booted, or the boot fails and the system halts.

In terms of build specifications, the invariant being verified here is the equality of the installed image and the retrieved image. In the notation of Example 2.6, we are ensuring `blob==blob_` in the presence of an adversary that can modify the post-installation storage state `storage_s[1]`.

For more practical information on this example, see [2].

3.3 Example (Traditional package management). A typical package management scenario is that a user wishes to download and install a software release associated with a particular project name.

Context. A user retrieves a software distribution package from a remote repository.

Adversary. May modify data in transit and at rest on the remote repository. May impersonate the remote repository at the IP level.

Security goal. The retrieved package is the desired official release of the named project.

[20] https://datatracker.ietf.org/wg/scitt/about/
[21] https://www.cve.org/
[22] https://u-boot.readthedocs.io/en/latest/usage/fit/source_file_format.html

Abstractly, the risks associated with this security goal can be controlled as follows:

1. An identity service is used to bind the project name to an authentication method x (for example, issuing a DID).
2. At the social layer, a claim format is agreed upon that specifies
   - A commitment to a package artefact.
   - A release number.

   An x-authenticated claim in this format is considered to vouch that the artefact to which the claim commits is the stated release of the named project.
3. Upon retrieval, the client establishes a trust relationship with the identity provider, verifies the authentication data, and measures the retrieved artefact against the authenticated commitment. If any of these three validation stages fails, the retrieval fails and emits an explanatory log message.

Concretely these checks are implemented with some standard technologies.

1. In widely used package repositories, a package is accompanied by a commitment in the form of a checksum or hash (e.g. SHA256), and a PGP signature of that commitment. The signing public key can be attached to an identity service in various ways, for example by registration on a PGP keyserver or by association with a TLS certificate.
2. The claim format is defined as an OpenPGP message with Signature Type byte $0x00$ [7, §5.2.1] which provides for an interpretation where the signer is the 'author' or 'owner' of the signed artefact.[23]
3. Suppose that the signing key has been associated with an identity using an X.509 certificate (for example, by using the same keypair to sign the certificate request, or by publishing the public key on an appropriately website served with HTTPS). Trust in this identity is then derived from validating a certificate chain leading to a trusted root certificate authority (CA), whose certificates are installed in advance to form a trust anchor.

   The user wishing to have the highest possible 'liveness' guarantees for the identity verification must connect to an online service to check certificate revocation status.

3.4 Example (Secure boot). Microsoft's Secure Boot technology is based on essentially the same model as Example 3.3.[24] Thus the user wishes to boot a system whose boot image and hardware devices, as viewed from the context of UEFI firmware, are subject to 'approval' by the authority of the platform owner. The platform owner's identity is bound to a *platform key* by the fact of its enrolment in platform firmware, and its authority can be delegated by the enrolment of additional *key exchange keys*.

The 'approval' claim for an EFI executable image is encoded as an X.509 certificate appended to the end of the image. Trust in the 'identity' of keys enrolled in the platform firmware is implicit in the use of the platform.

This system has a number of limitations:

- The lack of a flexible identity service makes it difficult for users to take control of their boot environment by registering their own authentication material to vouch for boot programs. Instead, one must derive trust in practice from keypairs owned by Microsoft.[25]
- Because the system cannot necessarily be expected to connect to the Internet at a very early stage in the boot sequence, identity and claim revocation status must be stored directly in firmware configuration. Since firmware configuration must usually reside in an extremely constrained storage component, this leads to some difficulties not visible at the OS level.[26]

[23] https://www.rfc-editor.org/rfc/rfc4880.html#section-5.2.1

[24] https://uefi.org/specs/UEFI/2.10/32_Secure_Boot_and_Driver_Signing.html#firmware-os-key-exchange-creating-trust-relationships

[25] https://www.rodsbooks.com/efi-bootloaders/secureboot.html

[26] https://github.com/rhboot/shim/blob/main/SBAT.md

- The scope of secure boot is limited to measured device driver loading and handoff between platform firmware and an EFI application. It cannot make claims about the loading of platform firmware itself, or about the subsequent payload of the bootloader.

3.5 Example (Abstract integrity check). In the data model outlined in §2, a generic integrity problem can be expressed as follows:

Context. A user wants to compute a target t of a build B in several stages $S_1, \ldots, S_n$. At each stage, some inputs are consumed and some intermediate targets t' are computed.

Adversary. Between each stage, the (tampering) adversary may modify the value of the intermediate targets t'.

Security goal. The result of the computation is either the true value of t or $\perp$, indicating failure.

The probability of achieving the security goal can be increased by storing and retrieving succinct commitments to the intermediate values whose integrity may be compromised.

4 Conclusion

We have described a powerful and flexible declarative language in which the various early initialization stages for physical machines can be couched, even when few logical abstractions are available. When fully developed, this type of approach can facilitate both innovation in boot sequence design, by empowering developers, and security risk modelling, by defining explicit components which are prone to analogy and generalization and about which security claims can be made. Future work should pursue actual implementation of automated boot sequence solutions following the declarative paradigm, and discover commonalities between security contexts and relate them to boot definition structure.

The security considerations discussed here mostly apply to all software distribution patterns, and not only to bare metal machine initialization. We advocate for community initiatives to encourage developers to make more precise claims about their software artefacts, study the effect of composing security claims, and provide highly available and timestamped infrastructure to register them.

References

[1] Various authors. *Building the Future on Bare Metal*. Retrieved 2023-11-01. URL: https://www.openstack.org/use-cases/bare-metal/how-ironic-delivers-abstraction-and-automation-using-open-source-infrastructure.

[2] Nathan Barrett-Morrison. *Securing U-Boot: A Guide to Mitigating Common Attack Vectors*. Retrieved 2023-11-01. URL: https://www.timesys.com/security/securing-u-boot-a-guide-to-mitigating-common-attack-vectors/.

[3] Henk Birkholz et al. *An Architecture for Trustworthy and Transparent Digital Supply Chains*. Internet-Draft draft-ietf-scitt-architecture-04. Work in Progress. Internet Engineering Task Force, Oct. 2023. 44 pp.

[4] Henk Birkholz et al. *Detailed Software Supply Chain Uses Cases for SCITT*. Internet-Draft draft-ietf-scitt-software-use-cases-02. Work in Progress. Internet Engineering Task Force, Oct. 2023. 15 pp.

[5] Sharon Boeyen et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. May 2008. DOI: 10.17487/RFC5280.

[6] Krishi Chowdhary. *Proton Mail to use blockchain to verify recipient's email addresses*. 2023. URL: https://www.tomsguide.com/news/proton-mail-to-use-blockchain-to-verify-recipients-email-addresses.

[7] Hal Finney et al. *OpenPGP Message Format*. RFC 4880. Nov. 2007. DOI: 10.17487/RFC4880.

[8] Edgardo Gerck et al. "Overview of Certification Systems: x. 509, CA, PGP and SKIP". In: *The Black Hat Briefings* 99 (1997).

[9] Nick Johnson. *ERC-137: Ethereum Domain Name Service - Specification*. 2016.

[10] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519.

[11] Steven L Kinney. *Trusted platform module basics: using TPM in embedded systems*. Elsevier, 2006.

[12] Chris Lamb and Stefano Zacchiroli. "Reproducible Builds: Increasing the Integrity of Software Supply Chains". In: *IEEE Software* (2021). DOI: 10.1109/MS.2021.3073045.

[13] Adam Langley. *Revocation checking and Chrome's CRL*. 2012. URL: https://www.imperialviolet.org/2012/02/05/crlsets.html.

[14] Hamed Taherdoost. "Understanding cybersecurity frameworks and information security standards—a review and comprehensive overview". In: *Electronics* 11.14 (2022), p. 2181.

[15] Jiewen Yao and Vincent Zimmer. "Building Secure Firmware". In: *Apress: New York, NY, USA* (2020).