

# PowerVM LPAR Linux Secure Boot

Daniel Axtens, 2020-10-30

## Background/Goal

Users wish to ensure the integrity and providence of boot components (bootloader, OS kernel) during boot by requiring that they bear a valid cryptographic signature from a trusted party.

This has been widely adopted in x86\_64 and increasingly AArch64 through UEFI and UEFI Secure Boot, and is increasingly a requirement for security certifications.

We want PowerVM LPARs running Linux to also have a secure boot capability.

Power machines do not implement UEFI, and there are a number of fundamental incompatibilities preventing its easy adoption, including differences in executable format and existing host secure boot work. Therefore, there is a need to design and implement an alternative scheme.

Ultimately, we want to support a Bring Your Own Keys model where systems administrators can decide what keys are trusted. (This is the model UEFI supports.) However, initially we are designing a system where distro partners provide keys to IBM and those keys are placed in firmware as a complete and immutable set of trusted keys.

## Appended Signatures

This scheme uses appended signatures heavily. Appended signatures were originally used to sign Linux kernel modules. An appended signature is a PKCS#7/CMS signedData message with some metadata and a ‘magic string’ that is appended to the end of a file. The hash is calculated over the entire contents of the file (as opposed to Authenticode).

## Threat Model

We consider an attacker who wishes to subvert the boot process and run code of their own choosing in the bootloader or kernel context. (An example of such an attacker is someone deploying a “bootkit” or a persistent rootkit.)

We consider an attacker with root privileges on the target machine, but without access to any trusted key material.<sup>1</sup>

As Linux’s kexec functionality can load another kernel, we also consider ensuring the integrity of the running kernel against a user with root privileges to be in scope.

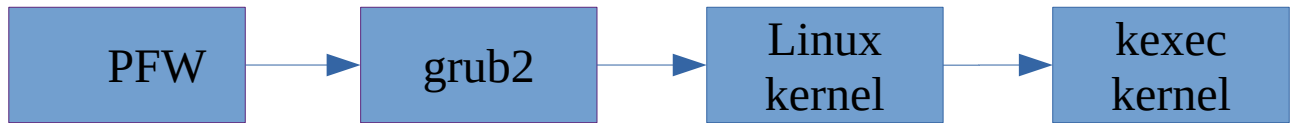
We consider attacks on the hypervisor to be out of scope.

---

<sup>1</sup> We have also considered the situation where there was a theft of key material or a range of signatures needed to be revoked (like the BootHole vulnerability). New keys would need to be provided to IBM and a new firmware release issued.

# Current Boot Process

The current boot process works as follows:



- Partition Firmware – the first firmware run in the LPAR context – is loaded by the hypervisor. PFW is trusted by virtue of the host’s firmware secure boot.
- PFW loads grub.
  - Grub is not signed or verified.
- Grub loads Linux.
  - Linux is already signed with an appended signature.
  - Grub does not verify this signature.
- Linux may boot a further kernel with kexec (for example for kdump).
  - This kernel is signed with an appended signature.
  - Linux does not verify this signature.

## Proposed Changes for Secure Boot

We propose the following changes.

### PFW to Grub transition

PFW needs a way to verify Grub. As we are already using appended signatures for Linux, and as the existing Power firmware secure boot capsule format would require a completely different set of key formats and systems, we use appended signatures to sign grub as well.

The technical details are provided in the appendix.

A distro partner signs grub with a key under their control using existing appended signature tools, and provides the public part to IBM to be built into PFW. Multiple concurrent signatures are possible, allowing for key rotation – grub will be loaded if any signature validates.

PFW will only enforce the signature requirement if booted in Secure Boot Enforce mode.

### Grub to Linux transition

Linux is already signed with an appended signature for Host Secure Boot. Grub will verify appended signatures on Linux against x509 certificate(s) embedded in grub.

- We teach grub how to parse PKCS#7 messages and extract the signature.
- We teach grub how to parse x509 certificates for public keys and use Grub’s existing RSA and SHA support to verify an appended signature.

- We use grub's existing verifier infrastructure to hook into the loading of Linux and other files that could compromise the integrity of Grub.
- Grub reads a Device Tree property to determine whether it should enforce appended signature verification.

In terms of key material, any number of keys can be baked into the grub image at grub-mkimage time. A kernel that has a valid signature by any embedded key is allowed to boot.

## Linux boot changes

To ensure continued kernel integrity (which is required to prevent kexec-ing into untrusted kernels), Linux will detect the presence of the device-tree property signifying secure boot and enter Lockdown integrity mode if it is present.

## Linux to Linux (kexec) transitions

To prevent kexecing into untrusted kernels, Lockdown requires that kexec kernels are signed by a key trusted by IMA. We are particularly concerned about kdump kernels in this instance.

To be able to kexec to the kdump kernel, the public key used for signing the kdump kernel needs to be in the IMA keyring or the platform keyring. For UEFI secure boot, it's loaded into the platform keyring. However, there is currently no mechanism for the key material used by grub to verify the kernel to be exposed to the kernel. Eventually, when we add support for Bring Your Own Key, we will provide a system for this, but for now, to enable kexec, the kernel signing key needs to be loaded into the IMA keyring through another mechanism, such as this:

- Load the Secure Boot CA into the `.primary_trusted_keys` keyring via the `CONFIG_SYSTEM_TRUSTED_KEYS` property. We assume the key used to sign the kernel is signed by this CA.
- Enable `IMA_LOAD_X509`, which allows certificates signed by a key on the `.primary_trusted_keys` keyring to be loaded into the IMA keyring. Then set `IMA_X509_PATH` to provide a path to the signing key on installed file system. That key will then be loaded into the `.ima` keyring at boot and be used to appraise the kdump kernel.

## References and Further Reading

- Linux Plumbers talk
  - <https://linuxplumbersconf.org/event/7/contributions/738/>
  - <https://www.youtube.com/watch?v=IJUNxHnopH4&feature=youtu.be&t=510>
- Patch sets
  - <https://lists.gnu.org/archive/html/grub-devel/2020-10/msg00152.html>
  - <https://lists.gnu.org/archive/html/grub-devel/2020-10/msg00151.html>
  - <https://lists.gnu.org/archive/html/grub-devel/2020-10/msg00048.html>
  - <https://lists.gnu.org/archive/html/grub-devel/2020-08/msg00037.html>

- Multiple-signer discussions
  - <https://lists.gnu.org/archive/html/grub-devel/2020-10/msg00067.html>
  - <https://lists.gnu.org/archive/html/grub-devel/2020-09/msg00081.html>

## Appendix: Signing grub with an appended signature

### The PreP problem

Normally, the presence of an appended signature is determined by looking at the last bytes of a file and comparing them to the appropriate magic string.

However, grub is often loaded directly from the PReP partition, where it is stored not on a file system but as a set of raw bytes. This makes it difficult to determine if an appended signature is present: because there is no filesystem, there is no concept of a file, and therefore no ‘end of file’.

Because grub is an ELF binary, we are able to determine where the binary ends, however as appended signatures can be of varying length, we cannot simply ‘skip forward’ a known length to check for an appended signature.

We ideally do not want to just reach the end of the grub binary and ‘keep scanning’ until we find something or hit the end of the partition: not only is this potentially inefficient, if an administrator installs a binary with an appended signature and then later installs one without an appended signature without explicitly zeroing out the PReP partition, they will be presented with an “invalid signature” error rather than a “missing signature” error, which could be confusing.

Any solution to this issue also needs to work for situations where the grub bootloader is stored as a discrete file with a known end-of-file: for example CHRP boot-info.txt boots used for booting from a CD, and netbooting.

### Appended Signature ELF note

We propose an ELF note that stores the appended signature. This ELF note must be the last part of the file, so that the appended signature still has the properties of being the last thing in the file – allowing the use of existing tools to create or inspect it.

This allows PFW to determine authoritatively, with only reference to the ELF headers:

- whether or not an appended signature exists,
- the size of the appended signature, and
- the location of the appended signature.

### Full specification of the note field

The name field of the Appended Signature Note shall be "Appended-Signature"; the type field shall be 0x41536967 (the ASCII values for the string "ASig"). It shall have an alignment of 4 bytes.

The appended signature section shall be the last section in an ELF binary.

Its format is as follows, noting that it is parsed "back-to-front"

1. **padding** - 0-3 bytes - used to ensure that descsz is a multiple of 4. Placed at the start of the description rather than the end to ease signature validation.
2. **PKCS#7 Message:** This is a PKCS#7 message that must be a signedData message. It shall contain no embedded data (that is, it is detached). It shall contain no signed attributes. An implementation may ignore or reject any unsigned attributes. It shall contain no CRLs, and any embedded certificates shall be ignored. It must contain at least 1 signerInfo blocks.

### 3. Appended Signature metadata:

```
u8 legacy_params[3];
u8 id_type; // Must equal 2
u8 legacy_param4;
u8 padding[3];
be32 signature_length;
```

4. **Appended signature "magic text":** "~Module signature appended~\n" (28 bytes, last byte is 0x0a, no trailing null byte.)

Parsing and validating an appended signature goes "back to front":

1. Identify the section as an appended signature section and note the namesz and descsz. If namesz and the name are invalid, or if descsz is not sufficient to hold the fixed size elements (magic text and module signature structure), validation fails here.
2. Using the namesz and descsz, move to the `_end_` of the section and read back the length of the appended signature magic. If the magic does not match the expected magic, validation fails here.
3. Read back a `module_signature` struct from the start of the appended signature. If the parameters are not as expected - all legacy parameters equal 0 and `id_type` equals 2, validation fails here.
4. `sig_len` is stored in network byte order (big-endian). Validate that `descsz = ALIGN_UP(siglen + size of module signature struct + size of magic string, 4)`, otherwise the validation fails here. Note that the magic string does not contain a terminating NULL.
5. The signature data is the `sig_len` bytes of data immediately preceeding the `module_signature` structure. The signature is over all bytes from the beginning of the ELF file (`\7fELF..`) to the beginning of the signature, including the padding.
6. Validate the signature according to PKCS#7 rules. Iterate over all `signerInfo` blocks: if any contain a valid signature is made with a trusted key, then the ELF binary is trusted.

## Multiple Signatures

- **Multiple signatures made at a single point in time:** if one party wishes to create multiple signatures with different keys, all of which they control (for example for key rotation), this can be done easily with OpenSSL. The appended signature contains a PKCS#7 message which can contain multiple `signerInfos`, one per key. PFW will validate the binary if any signature passes.

- **Multiple signatures made at different points in time:** Because the appended signature is stored in an ELF note, the size is encoded in the ELF header. As the signature is made over the ELF header when the first appended signature is added, it is not possible to change the size to add another signature without breaking the first. Currently, as the keys are fixed, there is no use-case for this feature, but it can be supported in future by allocating more space for the PKCS#7 message than is needed. The parser will read only as much message as the ASN1 format claims exists: any trailing information is discarded. Therefore, space can be left for additional signatures to be added later.