Open Network Install Environment (ONIE) UEFI Secure Boot Proposal

Title	ONIE UEFI Secure Boot Proposal
Authors	Curt Brune <curt@cumulusnetworks.com>
Created	04/03/2017
Version	2

Table of Contents

1 Change History

Version	Changes	Name	Date
1	Initial Draft	Curt Brune	04/03/2017
2	Minor clarifications after initial review	Curt Brune	04/17/2017

2 Overview

This document describes a method for supporting UEFI Secure Boot on ONIE enabled platforms.

Note: At the time of this writing only the **x86_64** CPU architecture is being considered. While the principles described herein are generally applicable to other CPU architectures and firmware, the details are quite a bit different and are beyond the scope of this document.

Note: This proposal is based on the following specification versions:

- UEFI Specification 2.6
- TCG TPM 2.0 Library Specification

2.1 Security and Trust

At its core, the notion of security is firmly grounded in the concept of trust. One party trusts another party or entity to behave in a well defined and consistent manner.

In an ONIE enabled computing environment end users place trust in the following components:

- Hardware
    - CPU silicon
    - FPGAs and CPLDs
    - Boot Firmware
- Software
    - ONIE
    - Network Operating System Installers
    - Network Operating Systems

2.1.1 Root of Trust

Ultimately a core component of a system must be explicitly trusted in order to form a root trust. The root of trust provides the foundation upon which to build further trusted relationships between system components.

In an UEFI Secure Boot enabled system, the end user trusts the hardware vendor to deliver a system where the hardware and boot firmware (UEFI) are trustworthy. When the system boots and UEFI is running, the system is in a trusted state. This forms the root of trust in a UEFI system.

2.1.2 Chain of Trust

A computing system boots up by loading and executing a sequence of bootstrap software components. Each component activates additional resources and functionality until finally the entire operating system is loaded and functioning.

For a UEFI Secure Boot system, each software component, starting with the root of trust, must first authenticate the next component in the sequence before transferring execution control to it.

The currently accepted best practice for authenticating digital objects is to verify digital signatures. The signer uses her private key to sign the digital object and the verifier uses the corresponding public certificate to authenticate the signature. Since the verifier trusts the signer and the digital signature checks out, the verifier trusts the signed digital object.

The root of trust verifies a signature on the first stage component using an embedded public certificate from a trusted entity. After verifying the signature, control is passed to the next stage, which may itself contain additional public certificates for verifying subsequent components. In this fashion the “chain of trust” is perpetuated, transferring trust to the next component.

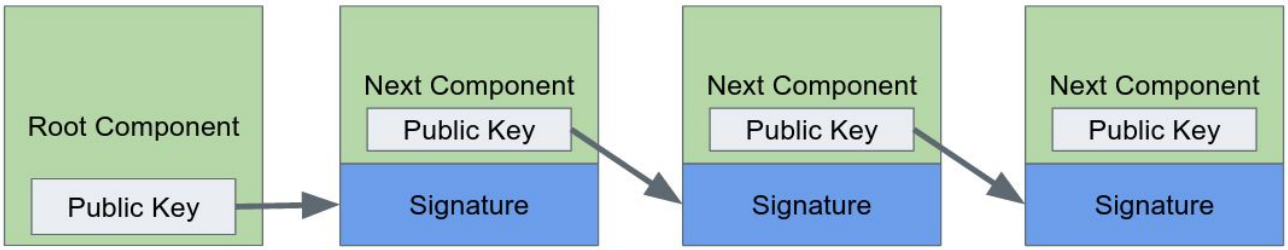

Figure 1. Transferring Chain of Trust

2.1.3 UEFI Image Authorization

In UEFI the firmware authenticates the first stage loader as described in UEFI Specification 2.6, section 30.5. To facilitate this the UEFI firmware maintains two databases, stored in UEFI environment variables:

- Authorized signature database (db)
- Forbidden signature database (dbx)

An image is authorized if at least one of the following is found in the authorized signature db:

- A matching cryptographic hash of the image
- A public certificate that can validate the image signature

Additionally, the image hash and signature must not be found in the forbidden signature database (dbx).

A de facto industry standard, adopted by almost all UEFI firmware vendors, is to include Microsoft's *Microsoft Corporation UEFI CA 2011* public certificate in the authorized signature database (db). If an image is signed by this certificate then the UEFI firmware will load and boot it.

2.2 Security and Measurement

In addition to trust, another axis of security is to "measure" the computing environment and compare the measurement to a known trusted environment. For modern computing platforms the Trusted Platform Module (TPM) provides hardware support for creating cryptographically verifiable measurements.

2.2.1 Measuring Objects

Measuring a single digital object is as simple as making a cryptographic hash (or digest) of the object. The well known `sha256` hash function is an example of a hash function:

```
digest = sha256(message)
```

To measure an entire environment (or state) a sequence of hash operations are concatenated together using the "hash extend operation":

```
New_Hash = hash(Old_Hash || message)
Old_Hash ← New_Hash
The || symbol stands for "concatenation"
```

On the next hash extend operation, the previous hash value is concatenated with the object and then the hash is taken of that entire bit sequence. In this way the hash can be extended indefinitely, while only requiring a finite amount of storage.

TPMs contain Platform Configuration Registers, whose purpose is to accumulate the results of hash extend operations. The PCRs are reset at system power on and can only be updated via hash extension. The trusted software components hash extend the computing environment into PCRs throughout the boot sequence.

For example, the boot sequence software hash extends the following components into PCRs during boot:

- UEFI environment variables
- First stage boot loader
- Second stage boot loader
- Operating System
- Loadable Operating System Modules

2.2.2 Attestation

Once the system is fully booted, TPMs support the ability to audit the PCR values in a cryptographically verifiable manner. This operation is known as attestation. An example workflow follows:

- A user process first issues a request to the TPM for the PCR values
- The TPM responds with a digitally signed copy of the PCR values
- The receiver verifies the TPM signature
- The receiver compares the PCR values to previously known trusted values

The signature verification and PCR value comparison is typically performed by a remote attestation server. If either the verification or comparison fails, the system should not be trusted.

3 Linux Shim and Secure Boot

The open source project “shim” is a common and generally accepted method for booting Linux in an UEFI Secure Boot environment. Many prominent Linux distributions use the shim project, including Redhat, Debian, Ubuntu and SUSE. See the “References” section for more details about shim.

The shim project consists of two EFI binary applications, shimx64.efi and MokManager.efi, discussed in the following sections.

3.1 shimx64.efi

shimx64.efi is a very thin EFI application that has the following properties:

- Small code base, making it easy to verify its correctness
- Typically signed by Microsoft Corporation
- Contains the shim owner’s embedded public certificate

Shim’s small code size allows for a quick security audit by the signing entity. In practice the signing entity is Microsoft. Having a small code size helps expedite the signing procedure.

The embedded public certificate plays an important role in continuing the chain of trust.

Since shimx64.efi is signed by a key in UEFI’s authorized key database, the UEFI firmware is able to authenticate shimx64.efi and load it.

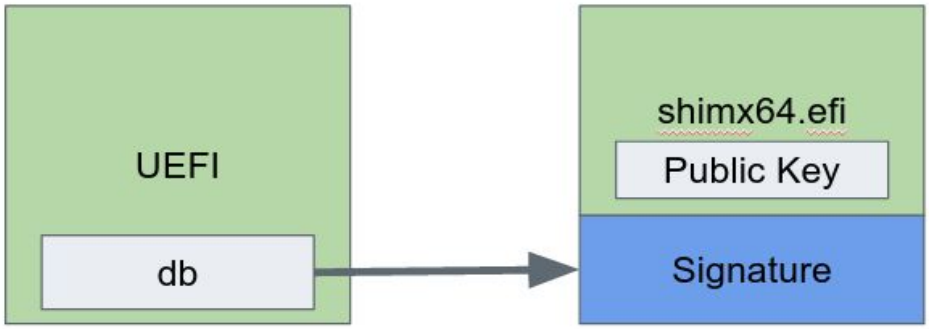

Figure 2. UEFI Firmware Loading shimx64.efi

Shim’s job is to locate, load and authenticate the next stage loader. For shim the next stage loader is grubx64.efi. grubx64.efi is an UEFI version of GRUB2, which is typically compiled by the same entity that compiles and distributes shim.

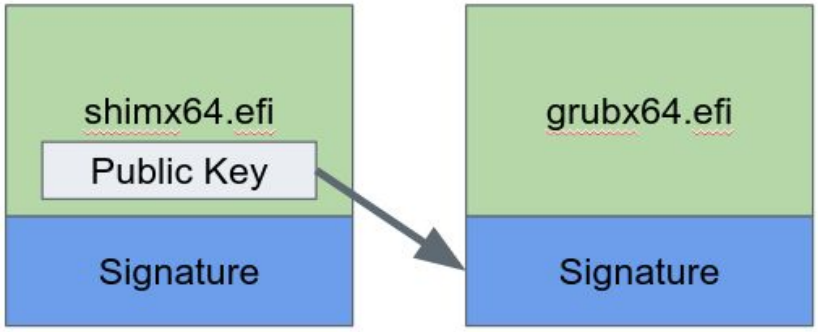

Figure 3. shimx64.efi Loading grubx64.efi

To authenticate the next stage loader, shim attempts the following:

- Use the same procedure as UEFI itself, i.e. validate the image using the db and dbx signature databases
- Use the Machine Owner Key (MOK) database managed by the MokManager, discussed in the next section.
- Use the shim owner's embedded public certificate

If shim is unable to verify the next stage loader, it defaults to launching MokManager.efi, allowing the machine owner to enroll her own public certificates for verification.

The salient point about shim is that Microsoft only needs to sign the shim binary. Microsoft does not need to sign the next stage loader (grubx64.efi) or the MokManager. Shim can continue the chain of trust by using the embedded public certificate to authenticate the next stage.

As a service for other UEFI applications, shim also registers a verification interface with the UEFI runtime. This interface provides the same authentication checks that shim uses for verifying the second stage loader. In practice this interface is used by grubx64.efi to verify the Linux kernel image before launching it.

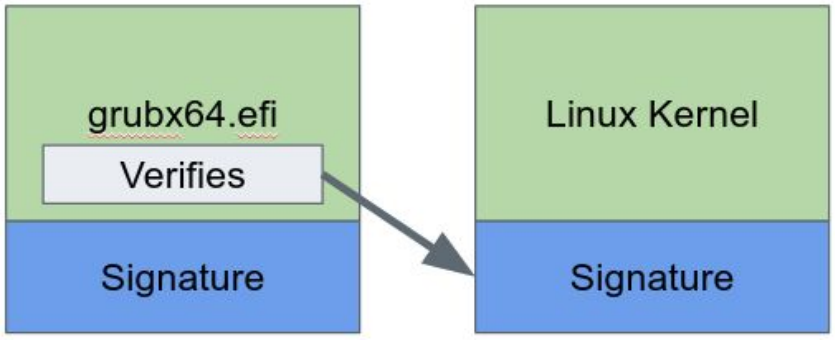

Figure 4. grubx64.efi Loading Linux Kernel

3.2 MokManager.efi

A Machine Owner Key (MOK) is a type of key, generated by an end user, used to sign EFI binaries of the user's choosing. Since both shim and GRUB (via shim's verification interface) consult the MOK databases, MOKs give the user the ability to load and execute locally compiled EFI binaries, such as kernels and boot loaders. This allows machine owners to run software of their choosing, while also maintaining the Secure Boot chain of trust.

MokManager.efi is an EFI application that allows the user to manage the MOK databases in a secure manner. This application is signed by the same key embedded into the shim application, thus shim is able to authenticate the MokManager.efi and load it.

The MOK system is comprised of two databases, similar to UEFI's `db` and `dbx` databases. These databases are named `mok` and `mokx` respectively. The `mok` database contains the authorized machine owner keys, while the `mokx` database contains revoked machine owner keys.

The MokManager.efi application provides a simple, interactive interface for administrating the MOK databases, with common operations such as:

- Add (enroll) MOKs into the mok and mokx databases
- remove MOKs from the mok and mokx databases
- clear and reset the mok and mokx databases

The mok and mokx databases are implemented as UEFI variables and can only be updated by the MokManager.efi application running under the chain of trust from shim. The contents of these database are "trusted" at the same level as the UEFI db and dbx databases. The databases are trusted because the only entity that can update them is signed and trusted.

3.3 Boot Sequence

The following diagram illustrates the complete Secure Boot sequence of a Linux kernel, putting together the pieces from the previous sections:

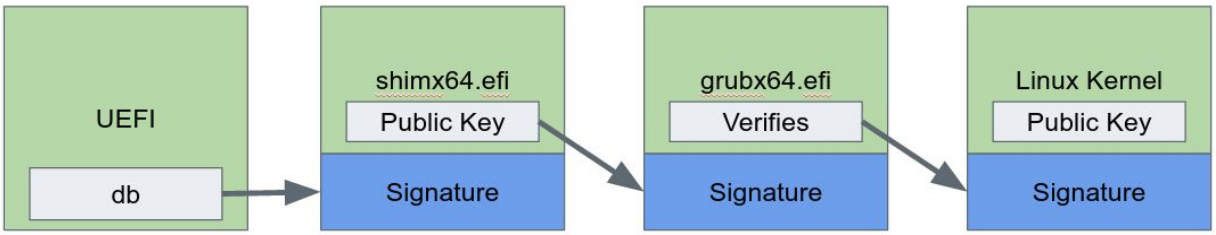

Figure 5. Booting Linux with Secure Boot

In words, the Linux kernel boot sequence proceeds through these stages:

- Power On
- Low Level “PRE UEFI” code measures (hash extends) UEFI firmware into TPM PCRs
- UEFI firmware verifies the signature on shimx64.efi and executes it
    - The public certificate of the entity that signs shimx64.efi must be in the UEFI authorized db.
    - Standard practice is that Microsoft’s public certificate is in the UEFI db and that Microsoft has signed shimx64.efi
    - Measures shimx64.efi into TPM PCRs
- shimx64.efi locates grubx64.efi, verifies its signature and executes it
    - Typically shim uses the embedded public certificate to verify grubx64.efi
    - Measures MOK environment variables into TPM PCRs
    - Measures grubx64.efi into TPM PCRs
    - shim also consults the UEFI db and the MOK database
    - If all else fails, shim attempts to load MokManager.efi
- grubx64.efi locates a Linux Kernel, verifies its signature and executes it
    - Uses the verification interface provided by shim, i.e. calls back into shim to verify the Linux Kernel signature using all the methods that shim uses
    - Measures Linux Kernel into TPM PCRs
- Linux Kernel boots
    - Can optionally continue the chain of trust and verify signatures on kernel loadable modules and user space applications

4 ONIE and Secure Boot

ONIE is a Linux based operating system. As such applying the shim Secure Boot model to ONIE is appropriate. This section describes the implications and changes required for ONIE to follow the shim model.

4.1 Applying the shim Model

Applying the shim Secure Boot model to ONIE requires additional software components that were not present in earlier versions of ONIE.

The hardware platform vendor builds and distributes ONIE along with the hardware. The additional software required for Secure Boot will also be provided by the platform vendor. This table describes the new components that the hardware vendor is responsible for providing:

Binary	Built By	Signed By	Contains	Notes
ONIE shimx64.efi	HW Vendor	Microsoft	HW Vendor public certificate	Measures, verifies and executes ONIE grubx64.efi
ONIE grubx64.efi	HW Vendor	HW Vendor	N/A	Measures, verifies and executes ONIE Linux kernel and initramfs
ONIE Linux kernel	HW Vendor	HW Vendor	[Optional] HW Vendor public certificate for verifying kernel loadable modules	

It is quite reasonable to expect that a single shimx64.efi binary can be shipped for all the x86_64 platforms of a hardware vendor. That minimizes the Microsoft code signing procedure to a single binary per hardware vendor.

The ONIE build process uses the hardware vendor's public/private key pair in the following manner:

1. The public certificate is embedded into the ONIE shimx64.efi binary
2. The private key and public certificate are used to sign the ONIE grubx64.efi binary
3. The private key and public certificate are used to sign the ONIE Linux kernel
4. [Optional] The private key and public certificate are used to sign ONIE kernel loadable modules

4.2 Installing ONIE

The end result of the ONIE build process is a UEFI compatible .ISO image that contains the above components plus a small amount of installer code. The .ISO image is suitable for installation via:

- A device reachable by the UEFI Device Path Protocol (e.g. USB)
- UEFI HTTP or PXE network based install

The installer code performs the following actions:

- Formats the disk for GUID Partition Table (GPT)
- Creates the EFI System Partition (ESP)
- Creates an ONIE partition
- Installs ONIE shimx64.efi, MokManager.efi and grubx64.efi into the ESP
- Installs the ONIE kernel and initramfs into the ONIE partition
- Configures GRUB2 to load the ONIE kernel and initramfs
- Modifies the UEFI **BootOrder** and **Boot####** global variables to boot into ONIE shim at the next boot.

In addition to the .ISO image, the ONIE build process also produces an ONIE updater image. The ONIE updater image, used for ONIE self-update, is suitable for running from within ONIE as a firmware update. Firmware updates are discussed in a subsequent section.

4.3 Using Signed ONIE Installable Images

With shimx64.efi and grubx64.efi in place, the ONIE Linux kernel can now boot securely. The next step is to extend the chain of trust to the ONIE installable images, also known as “installers”.

It is not strictly necessary to extend the chain of trust to the installers, but it is highly desirable. An untrusted (unsigned) installer could install an untrusted operating system, but that operating system would fail to load at the next boot because UEFI and shim would not be able to verify it. In this case Secure Boot did its job, protecting the system from untrusted software.

While that is true in theory, an untrusted installer could run amok and make life difficult in practice. An untrusted installer can do anything the root user in the ONIE kernel can do. Some examples of bad behavior:

- Spam the management network (eth0) with malicious traffic
- Delete disk partitions

Extending the chain of trust to the installers alleviates these types of problems.

4.3.1 Signed ONIE Installable Image Format

The term “ONIE Installable Image” describes image types that ONIE is capable of downloading and installing. These are the known ONIE installable image types:

- Network Operating System (NOS) installer -- installs a NOS
- Hardware Vendor Diagnostic OS installer -- installs a HW Diag
- ONIE updater image -- installs/upgrades ONIE
- Firmware updater image -- upgrades system firmware

The default ONIE installable image format is quite flexible. In order to support a signed installer, the format must be more rigorously specified.

The new format consists of these sections:

- Executable installer data
- Digital signature of the executable installer data
- Image Information Block

The “executable installer data” is the portion of the installer that the ONIE image discovery mechanism executes directly. This corresponds to the traditional ONIE compatible installer image format.

The digital signature covers the “executable installer data”.

The image information block (IIB) is a packed binary C-style structure, consisting of:

Field Name	Data Type	Notes
ONIE-Image-Id	128-bit GUID	Versions this structure.
Signature-Id	128-bit GUID	Identifies the signature type.
Signature-Offset	64-bit unsigned integer	Offset from the start of the image to the beginning of the signature, in bytes.
Signature-Length	64-bit unsigned integer	Length of the digital signature, in bytes.

> **Note:** The binary representation of the image information block fields is big endian (also known as network byte order). Specifically, the GUID binary representation follows RFC-4122, storing the data in network byte order. This is counter to the UEFI

Specification 2.6, Appendix A “GUID Format”, that specifies the little endian representation.

The ONIE-Image-Id for this structure is defined as:

```
ONIE-Image-Id := 216e9675-be17-46c7-aa71-e525eac83bd2
```

The ONIE image discovery mechanism can easily identify a signed ONIE installable image by attempting to read the ONIE-Image-Id GUID from the end of the image.

> **Note:** If the format of the image information block changes in the future, then a new ONIE-Image-Id GUID must be generated and used for that structure. The ONIE image discovery mechanism will look for all supported ONIE-Image-Id GUIDs.

At this time, the only supported signature type is PKCS#7, identified by the the UEFI GUID, `EFI_CERT_TYPE_PKCS7_GUID`:

```
EFI_CERT_TYPE_PKCS7_GUID := 4aafd29d-68df-49ee-8aa9-347d375665a7
```

> **Note:** PKCS#7 has been superseded by IETF RFC-5652. In the literature and tools, this is referred to as “Cryptographic Message Syntax” (CMS). The `openssl` tool sub-command `cms` supports this format.

Here is an example of an image information block.

Field Name	Value
ONIE-Image-Id	216e9675-be17-46c7-aa71-e525eac83bd2
Signature-Id	4aafd29d-68df-49ee-8aa9-347d375665a7
Signature-Offset	157286400
Signature-Length	1675

4.3.2 Signing the NOS Installer

The NOS vendor is responsible for preparing the NOS installer image in the required format, as described in the previous section. This includes:

- Signing the NOS installer with their private key and public certificate
- Preparing and appending the image information block

The ONIE project will provide a reference implementation of this procedure as part of the ONIE demonstration OS.

In addition, the NOS vendor must make the public part of their key pair generally available in form of a X.509 v3 certificate. This is required so that the end user may enroll the NOS vendor's public certificate into the MOK database. Without that, the hardware vendor's ONIE will be unable to authenticate the NOS installer.

4.4 Firmware Updates

Firmware, as defined by the ONIE project, includes:

- ONIE software (kernel + initramfs), everything in an ONIE updater image
- UEFI firmware. The data that lives in an 8MB SPI-ROM on most x86 platforms.
- CPLD programs. Most platforms have some number of CPLDs (3 seems typical) that require updates. CPLDs are typically upgraded via JTAG I/O signals, usually connected to a GPIO controller on the CPU complex.

Updating firmware with Secure Boot enabled may be accomplished with either the existing ONIE firmware update mechanism or with the UEFI Firmware Update Protocol. The UEFI Firmware Update Protocol is described in UEFI Specification version 2.6, section 22.1 and will not be described here.

An ONIE firmware updater image is a type of ONIE installable image created by the hardware vendor. The hardware vendor creates a signed ONIE firmware update image as described in Signed ONIE Installable Image Format. The hardware vendor signs the ONIE firmware updater image using their private key and public certificate.

Since the hardware vendor builds shim, which contains the hardware vendor's public certificate, the system has all the information necessary to authenticate the ONIE firmware updater image. It is unnecessary for the hardware vendor to load its public certificate into the MOK database, as was the case for the NOS vendor.

4.5 Hardware Diagnostic Operating System

The hardware vendor's workflow for creating the diagnostic operating system installer closely follows that of creating the ONIE firmware updater.

The hardware vendor creates the diag installer and signs it with their private key and public certificate.

Since the hardware vendor's public certificate is already embedded in shim, the system can authenticate the diag installer without consulting the MOK database.

4.6 Build System

Supporting Secure Boot and TPM access requires updates to the ONIE build system, including new tools and operations at build time and additional software included in the runtime ONIE image.

The following new software and tools are required for supporting Secure Boot and TPMs:

Software	Version	Use	Notes
pesign	0.112	Build time	Utility for signing PE/COFF executables
shim	0.9+	Runtime	Provides shim and MokManager
mokutil	latest	Runtime	User space MOK utility
tpm2 tss library	latest	Runtime	TPM2.0 libraries
tpm2 tools	latest	Runtime	TPM2.0 tools
efivar	latest	Runtime	Library for interacting with EFI variables
efibootmgr	latest	Runtime	Utility for managing EFI boot order
openssl	1.0.2k	Runtime Build time	Private / Public key swiss army knife
libnss3-tools	latest	Build time	More certificate tools
keyutils	latest	Runtime	Tool for managing Linux kernel keyrings
sbsigntool	latest	Build time	Utility for signing Linux kernel

4.7 PKI – Managing Keys and Certificates

While the generation of private/public key pairs and X.509 certificates is straightforward, the complete PKI lifecycle management of keys and certificates is beyond the scope of this document. The topic and terminology is a bit dense, with many closely related terms used interchangeably in various documentation and contexts.

Note: The key management techniques discussed in this section form a proof of concept example and are not suitable for production use.

For the purposes of this document, the recommendation is to follow the procedure outlined in Microsoft's Windows 8.1 Secure Boot Key Creation and Management Guidance document. The shim project also follows these recommendations.

Specifically, the requirements for ONIE PKI:

Object Name	Object Type
Public / Private Key	2048-bit RSA
Message Digest Algorithm	SHA-256
Signature Algorithm	SHA-256 with RSA Encryption
Signature Format	DER encoded CMS
Public Certificate	X.509 v3

All the examples in this section use the `openssl` command line utility.

4.7.1 Generating a Key Pair

This is an example of creating a 2048-bit RSA private/public key pair.

```
linux:$ openssl genpkey -out secret-key.pem -outform pem \
                -algorithm RSA -pkeyopt rsa_keygen_bits:2048
```

Note: The output file "secret-key.pem" contains both the private key and public key. This file must be kept private.

4.7.2 Extracting Public Key from Key Pair

This is an example of extracting the public key from the private/public key pair. The public key is available in "public.pem".

```
linux:$ openssl rsa -in secret-key.pem -pubout \
                -out public.pem -outform pem
```

4.7.3 Generating a Self-Signed X.509 Certificate

This is an example of generating a self-signed X.509 certificate. A self-signed certificate is useful for development and testing. In practice the X.509 certificate should be signed by a trusted certificate authority (CA).

```
linux:$ openssl req -x509 -new -outform pem -out public-cert.pem \
         -key secret-key.pem -keyform pem -sha256 -days 365
```

The request command will prompt for various fields of identifying information, such as:

- Country Name
- State or Province Name
- City Name
- Organization Name
- Organizational Unit Name
- Common Name
- Email Address

This is an example of dumping the public certificate in human readable form:

```
linux:$ openssl x509 -in public-cert.pem -text -noout
```

4.7.4 CMS Signatures

This is an example of creating and verifying a CMS (aka PKCS#7) signature.

First, create a sample document:

```
linux:$ echo “Hello World” > message.txt
```

Next, sign the document using the private key and the public certificate. The public certificate binds additional identifying information to the signature.

```
linux:$ openssl cms -sign -binary -in message.txt -outform pem \
        -out message.txt.sign -signer public-cert.pem \
        -inkey secret-key.pem
```

The creates the signature in the “message.txt.sign” file, encoded in PEM format.

Finally, verify the signature.

```
linux:$ openssl cms -verify -inform pem -in message.txt.sign \
        -content message.txt -CAfile public-cert.pem
Hello World
Verification successful
```

5 References

5.1 Open Network Install Environment

- https://github.com/opencomputeproject/onie
- https://github.com/opencomputeproject/onie/wiki

5.2 UEFI Specifications

- http://uefi.org/specifications

5.3 TCG TPM Specifications

- TPM Library Specification – https://trustedcomputinggroup.org/tpm-library-specification/
- PC Client Specific Platform Firmware Profile Specification – https://trustedcomputinggroup.org/pc-client-specific-platform-firmware-profile-specification/
- PC Client Work Group PC Client Specific TPM Interface Specification (TIS) – https://trustedcomputinggroup.org/pc-client-work-group-pc-client-specific-tpm-interface-specification-tis/

5.4 SHIM Boot Loader

- https://github.com/rhinstaller/shim
- http://www.rodsbooks.com/efi-bootloaders/secureboot.html#shim
- https://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/UEFI_Secure_Boot_Guide/sect-UEFI_Secure_Boot_Guide-What_is_Secure_Boot-Microsoft_Implementation.html

5.5 Digital Signatures

- https://en.wikipedia.org/wiki/Digital_signature
- https://technet.microsoft.com/en-us/library/dn747883.aspx
- https://tools.ietf.org/html/rfc5652

5.6 shimx64.efi Code Signing

- https://msdn.microsoft.com/en-us/windows/hardware/drivers/dashboard/uefi-firmware-signing

- https://blogs.msdn.microsoft.com/windows_hardware_certification/2013/12/03/microsoft-uefi-ca-signing-policy-updates/

6 Glossary

6.1 ACPI

Advanced Configuration and Power Interface.

6.2 CMS

Cryptographic Message Syntax, RFC-5652. Formerly PKCS#7 version 1.5, RFC-2315.

6.3 DER

DER is a binary format for data structures described by ASN.1. X.509 certificates are described by ASN.1 and can be encoded in DER.

6.4 GPT

GUID Partition Table.

6.5 GUID

Globally Unique Identifier. Synonymous with UUID. See RFC-4122.

6.6 IIB

Image Information Block -- Part of the ONIE installable image format.

6.7 MOK

Machine Owner Key.

6.8 PCR

Platform Configuration Registers, contained within a TPM.

6.9 PEM

A Base64 ASCII representation of DER.

6.10 PKCS

Public Key Cryptography Standards.

6.11 PKI

Public Key Infrastructure

6.12 TCG

Trusted Computing Group.

6.13 TPM

Trust Platform Module.

6.14 UEFI

Unified Extensible Firmware Interface.

6.15 UUID

Universally Unique Identifier. Synonymous with GUID. See RFC-4122.

6.16 X.509 v3 Certificate

An IETF standard that defines the format of public key certificates. See RFC-5280.