Configuration from bare metal to the cloud - leveraging modern systems to enhance manageability

Michael A. Rothman, *Intel* Vincent Zimmer, *Intel*

Abstract

Most users of a platform see the operating system as a fully instantiated entity. But from a system administration perspective, the platform is just as much the operating system (OS) and its application suite as it is the underlying substrate upon which the OS has been installed. And it is this substrate which includes hardware and firmware elements. Of the two, the firmware works in conjunction with the hardware and the OS-level installers and infrastructure code to both provision, manage, and recover a platform. The details of these firmware behaviors for management of the overall platform include capabilities to explicate the platform state, health, and security posture. Some of the platform capabilities can in turn be updated by the platform owner for interfaces that adjust the configuration of the machine.

To particularize some of the points made above, industry standard firmware infrastructure, such as found in the Unified Extensible Firmware Interface (UEFI) [1], shall be used to demonstrate user cases and scenarios around the bare metal secure management of a platform. This will include both the extant challenges and emergent opportunities in fusing these firmware capabilities with an overall management strategy.

1. Introduction

Managing platforms today can be rather chaotic especially when we consider that an administrator may need to deal with hardware from different Original Equipment Manufacturer (OEM)s. These platforms range from embedded, to mobile, handheld, desktop, and even servers. Each of these platform's configuration schema may differ, and each platform may have third party add-in devices, all of which are configurable (possibly in differing ways).

Add-in device vendors are also challenged, because each of them may get a request for quote (RFQ) by an OEM to comply with their manageability schema/methodology. This leads to the vendor having to have multiple SKUs of their own device to cover the manageability needs for their clients.

Because of some of the complications associated with getting all of the configurable devices to expose things in a well-understood way, managed aspects of systems were relegated to configurable options based on the platform's CPU, chipset, and other baseboard configurable options.

In a bare-metal world the agent in charge of the target has been the platform's BIOS (and in some cases, the BIOS would cooperate with a management controller, if available.) The features that had been offered in the BIOS that assisted the platform's administration were limited and certainly inconsistent.

Given that, the underlying BIOS had largely been responsible solely for the initialization of the platform hardware and ultimately launching a target software target (e.g. operating system). Over the last decade, the BIOS has evolved with the hardware and the previously arcane interfaces have now been standardized within a public forum known as UEFI [1].

It should be noted that in the past, the only platforms that had built-in connectivity were the special SKU'd platforms with a management controller. Platforms in the past rarely had a network card, and those that did had very limited use of the device prior to the OS being run.

Today's platforms almost always have some form of built-in network connectivity hardware, and as the hardware has evolved, so has the underlying BIOS to natively support it via standard programmatic interfaces.

In addition to exposing networking connectivity within this evolving BIOS infrastructure, added measures were enabled to expand the configurability of all aspects of the platform hardware, both motherboard-based as well as add-in devices. With this drive for standardization and facilitating standard programmatic interfaces to interact with the hardware in a bare-metal manner, the need for security would certainly be needed. Whether this was allowing a platform to load only properly authorized content, or enforcing the ability to enable secure policy handshakes from the moment the machine is powered on, to the launch of the OS, and up to and including running content within the OS – a chain of trust is now something that can be maintained. This chain of

trust is capability which would have been a serious challenge in previous years on reasonably open hardware platforms.

1.1 Pre-OS / OS Interaction

During the platform initialization, the BIOS is primarily responsible for initializing sufficient hardware to be able to launch the software target (e.g. Windows, Linux, etc.)

During this initialization, there are a variety of standard programmatic interfaces that are made available as well as static structures which enable interaction with the platform and expose information about the platform configuration. A simple illustration of this interaction is illustrated in Figure 1.

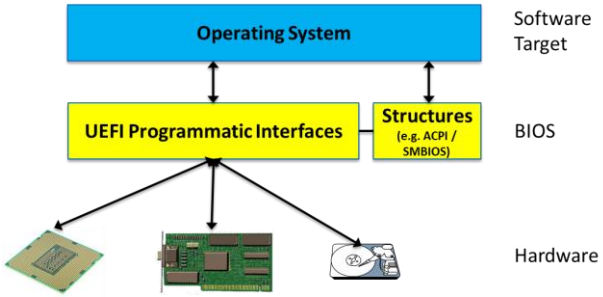

Figure 1

It should be noted that although this paper will focus on the capabilities exposed at the platform level due to standardization efforts within the BIOS, there are also significant new opportunities for synergy between the pre-OS environment and the OS and middle-ware layers.

Subsequent sections of this paper will further illustrate details on how bare-metal manageability, connectivity, and security are achieved within this pre-OS domain.

1.2 Exposed Capabilities

Even though there are thousands of pages of standards-based documentation detailing all aspects of the UEFI standards, this paper will attempt to highlight some aspects that will facilitate enabling bare-metal features such as provisioning, managing and recovering a platform.

1.2.1 Human Interface Infrastructure (HII)

During the platform initialization, one of the earliest facilities that is exposed is something called the HII Database. As Figure 2 illustrates, this database serves as a repository for all things related to the platform configuration and user interaction. This includes data such as user presentation forms (BIOS/device setup pages), strings (possibly encoded in many languages), keyboard layouts, fonts, etc.

Since this database is a central focus for lots of information that pertains to the platform's configuration data, it also holds much of what becomes central to establishing the necessary metadata for understanding how a platform is configured in a programmatic way.

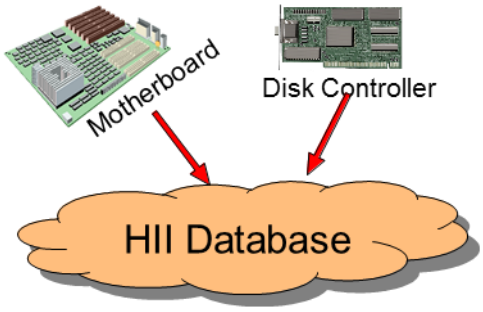

Figure 2

Much more of this will be covered in section 3 of this paper which will provide details of the configuration infrastructure and how this metadata which is contained within the HII database is used.

1.2.2 Networking

Platform firmware has had integrated network boot capabilities since the Pre-Boot Execution Environment (PXE) [1] was defined in the mid 1990's as part of the Wired For Management (WFM) [17] efforts. PXE can be thought of as 'in-band' networking since it runs on the main host CPU's, not an out-of-band chipset or platform microprocessor, or a 'non-host processor' (NHP). PXE entails two elements – (1) client side API's, including Base-Code and UNDI, for purposes of orchestrating the download of a network boot program (NBP) and (2) a wire protocol for interacting with the boot server. The PXE boot process is 'client initiated' in that the in-band firmware initiates a DHCP discovery process to start the networking interactions with a boot server.

PXE was originally part of the PC/AT BIOS and a specification jointly owned by a small consortium of companies. With the advent of the Extensible Firmware Interface (EFI) in the late 1990's, the 'base code' and 'UNDI' interfaces from BIOS PXE were mapped into EFI interfaces. This continued through the EFI 1.10 specification in 2001. The EFI1.10 specification was an Intel-owned document. In order to support broader industry adoption, EFI 1.10 was contributed to the Unified Extensible Firmware Interface (UEFI) forum in 2005, along with some post EFI1.10 networking API's. The latter included a modular IPV4 network stack that broke out IP, UDP, TCP, DHCP, ARP and other ele-

ments into separate API's, as opposed to the EFI1.10 reference implementation of the monolithic PXE stack. With this modular network stack in UEFI2.0 in 2006, the foundation was laid to create additional networking services on the UEFI platform. These services have included a refactored PXE Client that leverages the modular network stack and an ISCSI initiator, both in the open source.

More information on UEFI can be found at [5] and the specification itself at [1].

The question was posed to the industry group in 2007 about how to evolve PXE. At the time, there were many extant scenarios built upon the IPV4 PXE wire protocol, including support in all of the Linux distributions and the Windows Deployment Services (WDS) feature in Microsoft Windows. The most important feature request entailed addition of IPV6 support. As such, the UEFI Forum worked w/ the IETF and generated RFC 5970 that includes the option tags for IPV6 network boot. This RFC, along with a network interaction flow, form 'netboot6', or a variant of PXE that interoperates across IPV6.

The IETF and the UEFI Network Subteam (UNST), chaired by Vincent Zimmer [7], evolve the pre-OS wire protocols and UEFI API's, respectively.

The modular network stack and the IPV6 and IPV4 variants of PXE [9] can be found in the EFI Developer Kit 2 (edk2) project on source forge in the Network Package (NetworkPkg) [6]. Some details of the packages can be found in Figure 3.

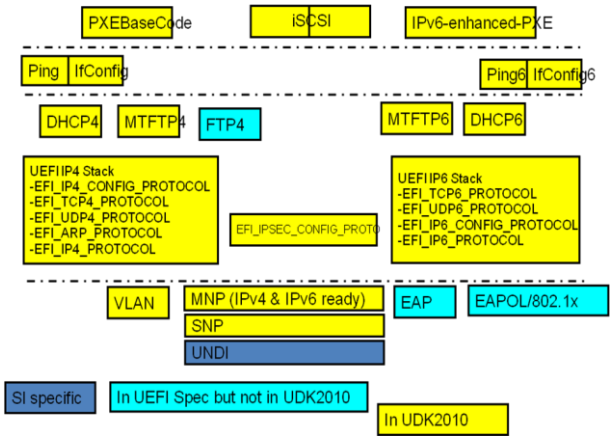

Figure 3 Network stack layout

Netboot6, along w/ UEFI Secure Boot [8], provided UEFI-only features not available on a PC/AT BIOS. These features, along w/ fast-boot, were integrated into Microsoft Windows8 and helped motivate the decision to mandate UEFI 2.3.1c specification conformance for this operating system in 2012. So the EFI effort that commenced in 1998 resulted in the 2012 launch of Windows8 that required UEFI.

This was the tipping point for the standards adoption and many of the preceding tactics, such as feature creation and open source, helped to motivate this decision.

1.2.3 Security

The management of a platform, especially across a network, in an enterprise network necessarily has considerations regarding platform security. To that end, there are a series of specifications and capabilities that describe the trust. This includes the underlying firmware infrastructure leading up to the UEFI interfaces. The former required some cryptographically signed updates, such as required by NIST 800-147. These updates should only be done under the authority of the Platform Manufacturer (PM). The latter portion of the platform, such as the extensible 3rd party UEFI executables, including the network boot program (NBP), are under the administrative control of the Platform Owner (PO). The PO administrative domain is often the same as the operating system or hypervisor administrator. In the case of UEFI executables, such as OS loaders, provisioning agents, and independent hardware vendor (IHV) drivers, cryptographic signing of the UEFI executables is applied.

In addition to authentication of the UEFI executables, the PM or PO may need to be authenticated for pre-OS operations, such as critical API usage, or during OS runtime. The former can be accomplished by the UEFI User Identity (UID) infrastructure, which allows for multifactor authentication. During OS runtime, there are few interfaces, with the most important for purposes of platform management that read on security concerns being the UEFI runtime API's of SetVariable, and UpdateCapsule. The former can be access controlled via read-only variables, but for art that must updated at runtime, the EFI_VARIABLE_AUTHENTICATED_WRITE can be used to ensure that only the creator of the variable can do field updates.

Finally, security considerations inhere in UEFI Pre-OS networking with network protocol authentication. There are two classes of network authentication, wherein the extant network infrastructure can challenge a new machine executing in the UEFI pre-OS when it attempts to join the network. For this case, the UEFI platform will surface some platform identity, such as via an 802.1x layer 2 challenge/response, prior to getting access to IP traffic. The latter case, wherein the UEFI platform wants to protect itself from possibly malevolent network or boot server infrastructure, is often handled via UEFI

Secure Boot wherein the platform will only execute remotely-delivered UEFI executables signed under one of the keys in the UEFI trust anchors, or the DB for UEFI Secure Boot.

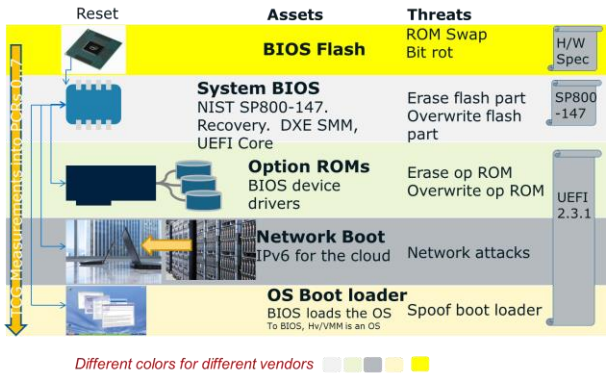

Figure 4

1.3 What To Expect In This Paper

Clearly within the scope of a relatively short paper, it is impossible to cover all of the details of any technology or even a semi-interest usage case. Since the subject matter of managing platforms is likely familiar ground for the reader, the authors want to expose advances in some of the manageability infrastructure.

Given that it is nearly impossible to keep up with the advances of all the manageability related technologies, this paper aims to focus the reader's attention on a few key elements that have recently evolved in the platform firmware environment.

After reading this, you should expect a better basic understanding of the advances in the pre-OS configuration and how that applies to managed platforms.

The paper will also cover the topics of both networking and security within the pre-OS environment, and explain what the recent advances have been within those domains and how they all relate to each other.

It is said that Socrates made a statement similar to this, "The only true wisdom is in knowing you know nothing."

The goal of this paper is to try and make the reader aware of certain advances that can be leveraged to enhance the manageability of platforms. Give them sufficient understanding so that further exploration by the reader is reasonably possible.

2. Pre-OS handshake

Historically, managed systems had at least in-band management capabilities – meaning that the target platform was running an OS with the necessary middleware for the administrator to remotely communicate with it.

In-band manageability didn't typically provide the ability to provision a system or significantly affect bare-metal operations such as BIOS settings or recovering from an issue prior to the launch of the OS.

In those situations, out-of-band (OOB) manageability would come into play. Platforms with OOB manageability would have a dedicated device such as a baseboard management controller (BMC) [2] whose job it was to enable communication between the client and the administrator, regardless of what may or may not be happening on the client's platform. This removed the dependency on the installed software actively running or in many cases, for the platform to even be on. The only requirement would oftentimes simply be that the managed target was connected to a power source (e.g. wall plug or battery.)

In many cases, systems can provide either methods of enabling a client-admin communication both, or even both. The communication illustrated in Figure 5 is a very typical example of a client-admin communication through a BMC which would typically run independently of whatever was occurring on the in-band platform processor.

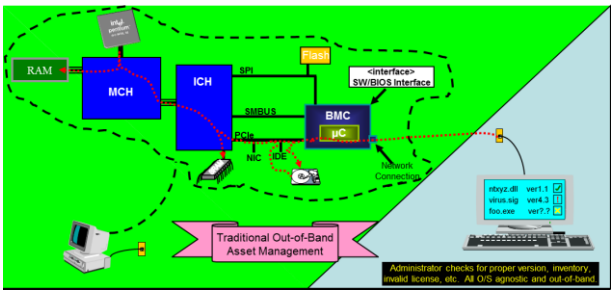

Figure 5

Given these methods of communication, there are data handshakes that occur within the platform in today's systems that had not truly occurred in the past. For instance, in figure 6 we demonstrate how third-party add-in devices within the system would communicate with the underlying BIOS interfaces. Prior to the advent of UEFI standards-based firmware, the configurability of add-in devices were limited to the user-initiated "Hit the Fx key to enter setup" and from an administrator perspective, these devices were largely black boxes. They either worked, or didn't work.

With the vast majority of OEMs and IHVs beginning to use the UEFI infrastructure, devices no longer have to be isolated as the black boxes they have been. Each configurable item in the platform has an opportunity to expose its metadata to the UEFI configuration infrastructure, and in turn will then be propagated to a variety of standard tables and interfaces which the OS would

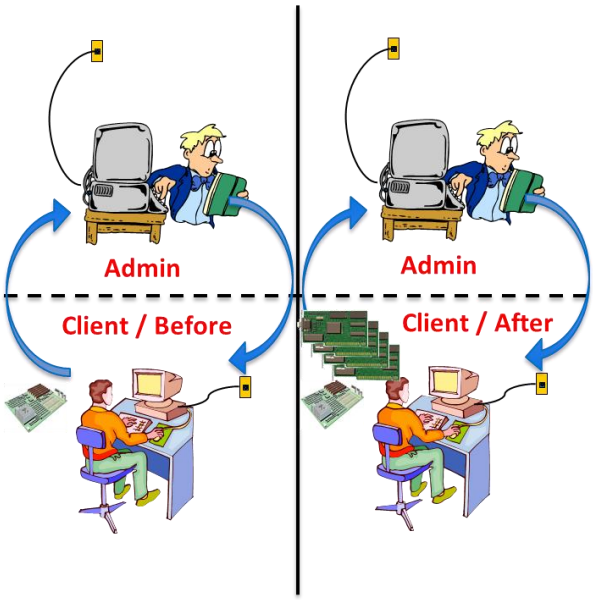

Figure 6

be exposed to. The left side of figure 6 illustrates the common scenario which has historically plagued managed systems. Much of the configurability of a system has been limited to soldered-down devices on the motherboard. On the right side of the illustration, there is a much more robust ability for third-party devices to be exposed and have a standard fashion by which they can be configured.

Whether this enhanced configurability is exposed via in-band or out-of-band manageability, the fact that such configuration portability exists enhances all manageability methodologies.

3. Configuration Infrastructure

So far, the paper has made reference to the existence of new capabilities to expose configuration data that had previously not been exposed.

This section will analyze the details associated with the UEFI configuration infrastructure, and exactly how it relates to enhancing the manageability of a platform.

3.1 Configuration Protocols

In the UEFI configuration infrastructure, there are several key protocols (interfaces) that exist which encompass how all user visible data is managed. Whether it is string information, forms, glyphs, or other extant data – there is a means by which such things are shared and exposed in UEFI systems today.

3.1.1 The Database Protocol

The EFI_HII_DATABASE_PROTOCOL interface is primarily responsible for the registration of data that encompasses configuration as well as usability elements such as fonts and images.

Any device that intends to be configured or in some way interact with the user will have to use the database protocol to achieve this. This paper will focus primarily on the forms and strings that get registered with the database and how they are used to achieve manageability goals.

3.1.2 The String Protocol

Once a package of configuration-related data has been registered with the database protocol, the EFI_HII_STRING_PROTOCOL interface can be used to simply extract the strings so that they can be used.

3.2 Localization

The concept of localization had historically been a limited subject in the BIOS world. With the advent of UEFI, the concept of string tokens was introduced. By tokenizing the strings, any device in the platform can register multiple versions of any given string.

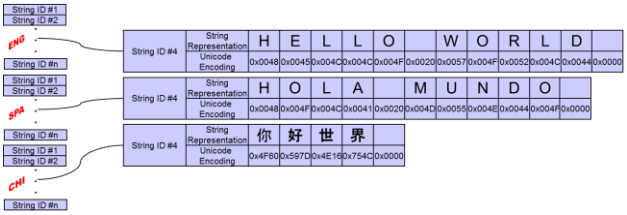

Figure 7

These different versions of string data were associated with a known language encoded by their corresponding RFC 4646 [4] language tag. Figure 7 illustrates how a single token (e.g. #4), could be interpreted as completely different elements based on the language tag specified when retrieving the string.

The language tags that are used for strings are generally associated with a language or language/country pairing. Examples being: es, en-US, de-DE. The first being Spanish (regardless of country), second being English as spoken in the United States and finally, German as spoken in Germany.

3.3 Forms

When registering configuration related data, there are two required elements, forms and strings.

Forms are binary encodings which define configurable objects, and one of the simplest examples of such a thing would be to think of a form being a web page. For each item on the page, there is a corresponding binary encoding which represents some configurable data.

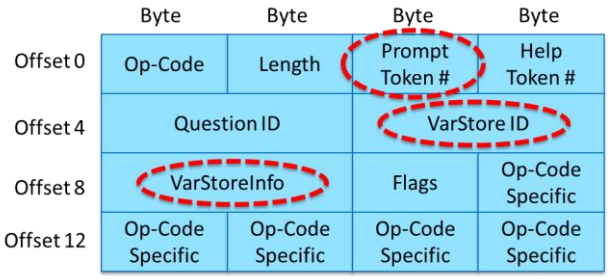

Figure 8 – Configuration question encoding

Figure 8 illustrates what a typical configuration question would look like. In each question, there are a few key items in the encoding that should be noted:

- Prompt Token #: The string token associated with the configuration question.
- VarStore ID: The ID associated with the variable storage declaration. This corresponds to where a particular configuration question's data resides on the platform. For instance, a particular variable storage declaration may correspond to a UEFI NV variable with a particular name (e.g. "Setup") and a particular GUID value.
- VarStoreInfo: This value would typically relate to the offset within the VarStore associated with a question. For instance, the offset value might be 23. So this particular question's current setting can be determined to be within the "Setup" UEFI variable at offset 23 into it.

From the encoding of the configuration op-code in Figure 8, it is easy to see how string references are intimately tied to each question. One thing that should be noted is that for each unique question, there is an associated Prompt Token.

We've already noted how each token value is a language agnostic reference to a string. For instance, Token #1 may have an English, Spanish and German equivalent. These are all languages that are human readable.

One fact that isn't obvious – but is leveraged for purposes of managing configuration elements is that not all of the language tags need to be human-readable. In other words, a device may expose a set of strings which are associated with a unique language tag (e.g. x-i-UEFI).

The "x" designation stipulates a private use language which the UEFI specification [1] uses as a UEFI_CONFIG_LANG_2 definition.

So if there is a forms-based question which asks, "Do you want to enable USB?" Let's assert that the PROMPT token has a value of 5. So if we probe the string protocol to determine what values token 5 have, we may certainly obtain one of them being the en-US value of "Do you want to enable USB?"

However, there may be a x-i-UEFI instance of token 5 which has a value of "USB_Enable".

OEMs typically have their own keywords and namespaces that they use when interacting with target platforms. Given that, a standard method to interact with a target platform might be one which leverages the syntax established by DMTF's SMASH CLP [3]. The typical CIM_BIOSAttribute associated with CLP expresses configuration data by using a "<OrgID>:<identifier>" syntax.

Given the previous example, one can imagine that a UEFI-based syntax could be expressed by having the x-i-UEFI language equivalent value replace the <identifier> value and the <OrgID> value would be UEFI.

So a CLP expression of the configuration question might resemble something like: "UEFI:USB_Enable"

3.4 Interaction with existing standards

When an administrator interacts with a UEFI compliant target, it would be trivial to request that the contents of the configuration infrastructure is exported. This means that the forms-based encoding and strings would minimally be exposed to the administrator.

If the administrator wanted to then enable USB on that particular system, they would search the string contents for the pertinent keyword (e.g. USB_Enable). Once the keyword is found, the administrator knows which token number the keyword is associated with and can then look in the forms data to see which question refers to that particular token.

After identifying which question the keyword is associated with programmatically, the administrator has sufficient information to know where that setting is stored on the system, and most importantly, how to change the setting.

This is a powerful concept because an administrator can now interact with a heterogeneous set of platforms that are all UEFI compliant and broadcast the same request to everyone without having to have special schema-based knowledge of each vendor's device.

4. Networking Infrastructure

So the IFR and HII mentioned above can convey elements that may be necessary for configuring the network infrastructure, such as the Network Interface Card

(NIC) MAC address. For that case, a local administrator may use a pre-OS UI to configure this information, or a UEFI protocol that accesses the out-of-band (OOB) management controller on the platform, may be used to get this information. A more common case, though, is that the in-band UEFI network will be used to download an agent that can locally modify the UEFI variables and interact with the UEFI protocols in order to update the UEFI variables.

One such topology is shown in the figure 9.

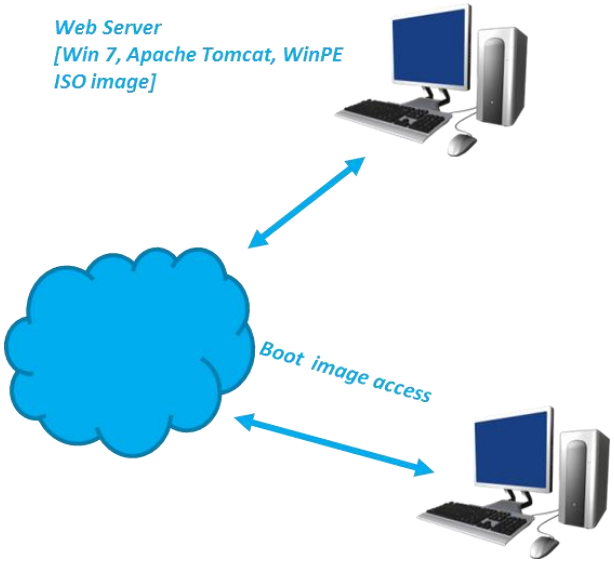

Figure 9 – Client / Server interaction

A more detailed view of this flow includes the below. The recovery action can include downloading a disk image, diagnostics, or other configuration tools. The download can be the PXE 2.1 based upon TFTP, with its connectionless UDP. With RFC 5970 and boot from URI, though, the boot server can be expressed as a URI. Recall that Netboot6 is a combination of the wire protocol defined in both RFC 5970 [10] and chapter 21.3.1 of the Unified Extensible Firmware Interface 2.4 specification [1]. The UEFI client machine uses DHCP as a control channel to expose its machine type and other parameters as it attempts to initiate a network boot. This is referred to as 'client initiated' network boot, as opposed to 'server initiated.' Examples of the latter include Intel(R) Active Management Technology (AMT) Integrated Disk Electronics Redirection (IDE-R), or exposing the local hardware network disk interface to the management console for purposes of the management control provisioning a disk image [11]. An implementation of Netboot6 can be found at in order to demonstrate a client-initiated download.
For client-initiated network bootstrap art like Netboot6, what are the details of the parameters? The most important parameter entails the architecture type of the .efi image that the boot server needs to provide. The client machine that has initiated the network boot needs to expose its execution mode to the boot server so that the appropriate boot image can be returned. Recall that UEFI supports EBC, Itanium, ARM 32, ARM 64, Intel 32-bit, and Intel 64-bit. This list may grow over time with corresponding updates to the UEFI Specification of machine bindings. Beyond a UEFI-style boot, some of my co-authors on 5970 worked for IBM and wanted to network boot a system software image over 1) HTTP and 2) not based upon UEFI technology. As such, the parameters at [12] cover both UEFI and non-UEFI, with the latter class including PC/AT BIOS and both PowerPC Open Firmware and Power PC ePAPR, respectively.

So RFC 5970 can be used in scenarios beyond Netboot6's TFTP-based download. This is enabled by the architecture type field extensibility, and also by the fact that the boot image is described by a URI, not a simple name with an implied download wire application protocol of TFTP as found in PXE2.1 IPV4 usages.
A way to explain this further can be done by examining our Linux configuration use case. In Linux, the DHCP server actions are performed by the dhcpd, or "Domain Host Controller Protocol Daemon." The daemon is parameterized by the file dhcpd.conf.
Within dhcpd.conf we enable Netboot6 by way of the following lines:

```
option dhcp6.client-arch-type code 61 = array of
unsigned integer 16;

if option dhcp6.client-arch-type = 00:07 {
  option dhcp6.bootfile-url
"tftp://[fc00:ba49:1625:fb0f::137]/bootx64.efi";
} else {
  option dhcp6.bootfile-url
"tftp://[fc00:ba49:1625:fb0f::137]/bootia32.efi";
}
```

The notable aspects are 'arch type' field and then the 'tftp' term. The bootx64.efi or bootia32.efi program, also known as the Network Boot Program (NBP), when executed on the local client (hopefully with UEFI Secure Boot logic applied prior to passing control into the image) can use any of the UEFI networking API's in the protocols defined in the UEFI Spec to download further .efi images, data files, or the operating system kernel. The device path protocol on the loaded image protocol of the NBP can be used by the NBP code's implementation to find the network address of the boot server from which the NBP was loaded, too.

As mentioned earlier, this technology isn't limited to a

UEFI style boot, though. A Linux PowerPC Open Firmware boot could be done with the same dhcp.conf by adding

```
if option dhcp6.client-arch-type = 00:0c {
  option dhcp6.bootfile-url
"http://[fc00:ba49:1625:fb0f::137]/linux-powerpc-kernel.bin";
}
```

to enable booting a PowerPC based native binary of Linux from a web server.

To leverage this flexibility of boot from URI, additional types can be added, as noted in [10].

5. Security Infrastructure

As mentioned earlier, for the network deployed platform configuration, there are two models. The first is where the network infrastructure protects itself from the candidate bare metal UEFI platform. In that case, the sysops may not even want the UEFI machine's network stack on its network. In that case, the network perimeter can be protected via a 802.1x controlled port. The associated challenge response protocol in that case is shown in figure 10.

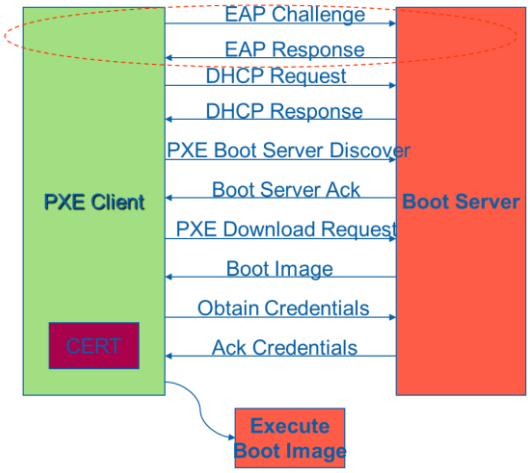

Figure 10

In this case, the UEFI machine, or the supplicant in terms of this topology, will provide some authentication to the server.

The 802.1x qualified challenge response is a precursor to the PXE download.

As a quick background:

Intel (R) Boot Guard binds the OEM low level boot firmware (PI code as exemplified by SEC/PEI/DXE) with the hardware, so the Boot guard trust anchors would not directly interface with the trust anchors for 3rd party UEFI content. Details on Intel Boot Guard can be found on page 4 of [13].

People often ask about the relationship of something like Intel (R) Boot Guard and its "Verified Boot" versus UEFI Secure Boot, as defined in chapter 26 of the UEFI 2.4 specification. Some background on the difference between platform and UEFI Secure boot can be found at [14], page 16. The "Reset Time Verified Launch" in Figure 5 of [14] logically maps to something like Intel(R) Boot Guard. The verification happens 'before' UEFI PI code and vets the provenance of that code, typically if the code was created and updated under the authority of the system board manufacturer. UEFI, on the other hand, is on the right hand side of that flow.

In other words, the underlying PI-code update key, say for validating a capsule update (install time verification) or the embedded signature of the PI code (load time verification) should not be the PK but some other system board vendor-managed key store. Recall that on certain x86 systems the end user could even edit the PK via a physically-present setup page. In that latter case, having the end user control the PI update key (and associated system firmware updates) is often not desired. In the PI specification there are definitions of signed firmware files and volumes, but there is no defined policy store and trust anchors for 'Secure Boot' of PEI and DXE elements.

In the end, users want end-to-end integrity, though, so both protection of the underlying firmware and the run time are important. This is shown in the figure 11.

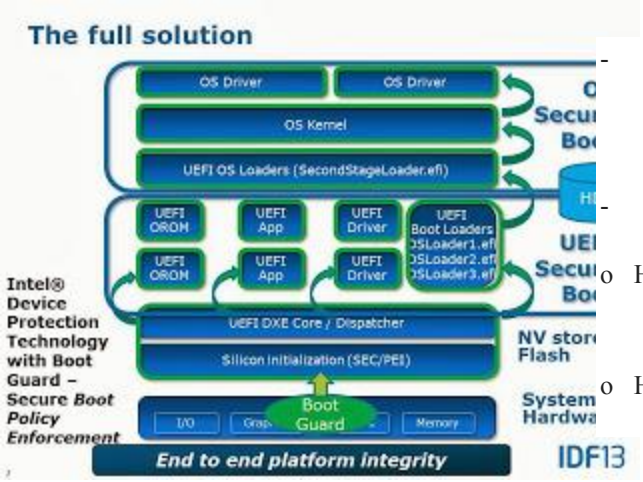

Figure 11

Note in this picture above that Intel (R) Device Protection Technology with Boot Guard surfaces from the system hardware and precedes execution of the PI SEC/PEI/DXE codes.

UEFI Secure Boot, on the other hand, is intended for 3rd party UEFI content, such as UEFI drivers or applications on the UEFI system partition. Intel(R) Boot Guard and PI code verification keys should have their own manifest and storage structure. For the 3rd party trust anchors, the place where this enrollment would happen is with the UEFI Secure Boot key hierarchy. The hierarchy for UEFI Secure boot includes the PK, KEK, DB, DBX. The factory-default configuration typically entails a PK that is owned by the OEM, and the PK authorizes updates to the KEK. The KEK is OS Vendor1 + OEM + other OS vendors, and the KEK entries authorize updates to the DB/DBX. DB is the 'allowed' list of code that can execute, and for a Microsoft (R) Windows8 machine contains a Microsoft OS certificate, the Microsoft UEFI CA cert, and possibly other OSV/ISV entries. Some of the theory/rationale behind this design can be found in [8].

Now for going from theory-to-practice-

Given a population of UEFI Secure Boot capable machines in the field, how is a pre-OS Independent Software Vendor (ISV) able to deploy content (i.e., the action item from above)? The short answer is that the ISV has 2 options:

- 1. Sign up w/ Winqual and get the UEFI driver/application signed by the UEFI CA

and/or

- 2. Create own verification certificate and
  - Have end user enroll manually

and/or

  - Have OEM preinstall (or update in field via firmware update)

An ISV can do 1+2 above since UEFI Authenticode-based executables support 'multisigning' so that they can be signed by BOTH the UEFI CA and the ISV's own key (see more on the final links below w/ SUSE example).

For the first option 1. above, the ISV can sign up w/ Microsoft Winqual and submit their content to be signed by the Microsoft UEFI CA. Most ISV's, IHV's, and non-MSFT OSV's already has a Winqual account if they deliver signed Windows drivers today since Microsoft has been doing kernel mode driver signing since Vista SP1. In addition, most IA machines that support UEFI2.3.1 Secure Boot carry a Microsoft UEFI CA DB certificate, so getting signed by the MSFT UEFI CA will mean that the ISV's .efi UEFI driver or application will simply work on a large class of UEFI machines.
For the second option 2. above, if the ISV wishes to generate its own roots and manually enroll in a PC (e.g., using PC setup screens) or distribute its keys for the OEM's to pre-enroll, some details on the process can be found at [6]
If you have a machine at home, you can use some of the flows described in the white paper above on running dmpstore and other commands at the UEFI shell to discover the configuration of UEFI Secure boot. A more user friendly way is to run the Secure Boot Checkup Utility [15] from Insyde on your Microsoft (R) Windows 8 machine.
This is the report from a Asus Windows 8 Intel (R) i3 touch laptop. The output from the report proceeds below:

Secure Boot Status on this system:
System Status: MS Required KEK: MS Required
OS Cert: 3rd Party (MS CA):
Secure Boot Enabled Present Present Present

UEFI Variables:
SetupMode:
SecureBoot:
OsIndicationsSupported:

BootOrder Item List:
BootCurrent: Boot00000 1 0000000000000001 0000 0000 Windows Boot Manager

Secure Boot Database Contents:

PK Variable Certificate (Platform Master Key):
X.509 Certificate:
CN=ASUSTeK Notebook PK Certificate

KEK Variable Certificates (Database Management):
X.509 Certificate: X.509 Certificate: X.509 Certificate:
CN=ASUSTeK Notebook KEK Certificate
CN=Microsoft Corporation KEK CA 2011
CN=Canonical Ltd. Master Certificate Authority

db Variable Certificates and Hashes (Allowed Signers):
X.509 Certificate: X.509 Certificate: X.509 Certificate: X.509 Certificate: X.509 Certificate:
CN=ASUSTeK Notebook SW Key Certificate
CN=ASUSTeK MotherBoard SW Key Certificate
CN=Microsoft Corporation UEFI CA 2011
CN=Microsoft Windows Production PCA 2011
CN=Canonical Ltd. Master Certificate Authority

dbx Variable Certificates and Hashes (Forbidden Signers):
X.509 Certificate:
CN=DO NOT TRUST - Lost Certificate

The interesting thing about this machine is that there is both a ASUSTeK KEK and Canonical KEK, along with the Microsoft KEK. So this set of KEK entries includes one for the OEM and two alternative operating system vendors, namely Microsoft for Windows and Canonical for Ubuntu [16].

Figure 12 is a friendlier view of the tool in action.

Figure 12

With the rich infrastructure of UEFI Secure Boot and the multi-tenant nature of the trust anchors in the DB, various IT and software entities can be represented in this trust relationship.

6. Platform Examples

The management usages described above can span a large variety of platforms, from phone to multiprocessor server.

So we have seen configuration, networking and security elements detailed. The UEFI infrastructure provides the substrate upon which these capabilities are deployed, but the capabilities did not speak to a specific class of platform. In fact, the management usages described above can span a large variety of platforms, from phone to multiprocessor server. Since all of these platforms have a host processor running UEFI style firmware, in addition to a networking capability that may be accessible for UEFI, the associated UEFI configuration objects can be leveraged to make these manageable, deployable platforms.

7. Related Work

The Desktop Management Task Force [3] and the Open Mobile Alliance [17] have manageability elements, but neither provide the combination of management interfaces, stylized binary configuration data, networking API's and security considerations as found in the UEFI work described herein.

8. Future developments

The infrastructure in this document can include a generic 'configuration language' to allow for seamless cross-vendor, cross platform migration, including machine cloning. In addition, the networking API's continue to evolve, as do the networking media, including 3G/4G

and Wi-Fi. Finally, security considerations are ever-present and the classes of security hardware, software access control mechanisms, and machine capabilities continue to evolve.

References

[1] UEFI Specification URL: http://www.uefi.org/specifications

[2] Usage of BMC in industry: http://www.dell.com/downloads/global/power/ps4q04-20040110-Zhuo.pdf

[3] DMTF Schema reference: http://schemas.dmtf.org/wbem/cim-html/2/CIM_BIOSAttribute.html

[4] Language identification tags: http://www.ietf.org/rfc/rfc4646.txt

[5] Zimmer, et al *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*, Second Edition, November 2010

[6] EFI Developer Kit II http://edk2.sourceforge.net

[7] Mark Doran, Vincent Zimmer, Michael Rothman, "Beyond BIOS: Exploring the Many Dimensions of the Unified Extensible Firmware Interface," in *Intel Technology Journal - UEFI Today: Boostrapping the Continuum*, Volume 15, Issue 1, pp. 8-21, October 2011, ISBN 978-1-934053-43-0, ISSN 1535-864X http://noggin.intel.com/technology-journal/2011/151/uefi-today-bootstrapping-continuum

[8] Magnus Nystrom (Microsoft), Martin Nicholes (Insyde), Vincent Zimmer, "UEFI Networking and Pre-OS Security," in *Intel Technology Journal - UEFI Today: Boostrapping the Continuum*, Volume 15, Issue 1, pp. 80-101, October 2011, ISBN 978-1-934053-43-0, ISSN 1535-864X http://noggin.intel.com/technology-journal/2011/151/uefi-today-bootstrapping-continuum

[9] Pre-Boot Execution Environment 2.1 http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf

[10] T. Huth (IBM Germany), J. Freimann (IBM Germany), V. Zimmer (Intel), D. Thaler (Microsoft), "DHCPv6 Options for Network Boot," Internet RFCs, ISSN 2070-1721, RFC 5970, September 2010, http://www.rfc-editor.org/rfc/rfc5970.txt

[11] Advanced Management Technology **http://software.intel.com/sites/manageability/AMT_Implementation_and_Reference_Guide/default.htm?turl=WordDocuments%2Fsetsoliderandotherbootoptions.htm**

[12] Boot Architecture Types http://www.iana.org/assignments/dhcpv6-parameters/dhcpv6-parameters.xml

[13] Intel CPU documentation **http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/4th-gen-core-family-mobile-brief.pdf**

[14] Platform Security http://uefidk.intel.com/sites/default/files/resources/Platform_Security_Review_Intel_Cisco_White_Paper.pdf

[15] Secure Boot Utility **http://apps.insyde.com/sbutil.html**

[16] Ubuntu **http://www.ubuntu.com/**

[17] Wired For Management http://www.intel.com/design/archives/wfm/downloads/base20.htm

[18] Open Mobile Alliance http://openmobilealliance.org/about-oma/work-program/device-management/