Cloud Net-Booting Beyond BIOS Using the Unified Extensible Firmware Interface

Vincent J. Zimmer
System Software Division
Intel Corporation
DuPont, Washington, USA

Abstract *- This paper describes a means by which a platform containing an implementation of the Unified Extensible Firmware Interface (UEFI) can be used for network booting, provisioning, and service-based scenarios in the pre-operating system (pre-OS). These scenarios include issues of handling scale, throughput, resiliency and trust. Beyond scale, UEFI-based networking supports a compatible path from today's net boot, moving the platform boot into IPV6-enabled world, and meets emergent compute models such as cloud computing .*

Keywords: Cloud, UEFI, IPV6, Networking, Trust

1 Introduction

The first several million instructions executed when a system is powered on are firmware, software stored in a chip. The firmware is responsible for initialization of much of the system, including important components such as RAM, video, and keyboards and mice. The firmware is responsible for finding and loading the operating system (such as Microsoft® Windows XP ® or Linux) from a number of different types of media, ranging from hard disks to LANs. The most challenging bootstrap scenario entails the network facing load scenarios. This challenge is driven by the distributed and possibly hostile nature various topologies. Firmware cooperates with infrastructure software in order to load the network boot program (NBP) such that it can to load further parts of the operating system before the operating system completely takes over. Today, on PC-class machines at least, that class of software providing this initial networking layer is known as "BIOS."

What do we mean when we mention "UEFI" and "Beyond BIOS?" "UEFI" [1] is an industry group that is standardizing what were the EFI1.10 specification and the Framework PEI (Pre-EFI Initialization) and DXE (Driver Execution Environment) specifications [3][4]. Within UEFI, the main UEFI specification (beginning with UEFI2.0) is handled by the UEFI Specification Working Group (USWG) and the Platform Initialization (PI) Architecture PEI/DXE content are handled in the Platform Initialization Working Group (PIWG). The differences are that USWG is focused on the OS-to-platform interfaces, whereas the PIWG is focused upon platform initialization. The USWG parties of interest are the OS (operating system) vendors, pre-OS application writers, and independent hardware vendors who write boot ROM's for block or output console devices. The PIWG parties of focus include the platform builders, such as Multi-National Corporations/Original Equipment Manufacturers (MNC's/OEMs), chipset vendors, and CPU vendors. The PIWG work can accommodate both UEFI operating systems and today's conventional/Int19h-based OS's.

PIWG-based components provide one means of producing the UEFI interfaces. What is common across both UEFI and PI Arch is the extensibility of code and interfaces. For UEFI and PI DXE, the extensibility point is a loadable driver model and for PI PEI extensibility is through firmware files with PEI Modules. Although it is common to think of firmware as being stored in a ROM (Read-Only Memory), most firmware is stored in NOR flash devices, which act like a read-only memory (ROM) but may be updated to add enhancements or fix bugs. The flash devices are divided into the equivalent of sectors on a disk.

1.1.1 Introduction

The emergence of more secure, scalable boot scenarios in the cloud entails updates to both the boot server and clients. On the boot server, UEFI firmware [1] that may be enhanced via additional trust [4] and isolation [5] properties allows for a hardened endpoint. In addition, though, the composition of wire protocols for machine authentication and end-the-end trust on executable and data download are needed. Beyond this, support for IPV6 is needed since platform hardware going forward are required to support this based upon either purchasing requirements [21] or to address paucity of IPV4 addresses [22].

1.1.2 UEFI PXE Protocol Extensions

UEFI PXE [18] extension set is defined on a foundation of industry-standard Internet protocols and services that are widely deployed in the industry, namely TCP [27], DHCP [4], and FTP [26]. These standardize the *form* of the interactions between clients and servers. To ensure that the *meaning* of the client-server interaction is standardized as well, certain vendor option fields in DHCP protocol are used, which are allowed by the DHCP standard. The operations of standard DHCP and/or BOOTP servers (that serve up IP addresses and/or NBPs) will not be disrupted by the use of the extended protocol. Clients and servers that are aware of these extensions will recognize and use this information, and those that do not recognize the extensions will ignore them.

Another feature of these download is that the server name and options can also be pre-provisioned and/or ascertained by the UEFI client in some out-of-band fashion. For provisioning a large set of machines in a datacenter, such as the Cloud, the use of PXE from bar-metal provisioning represents a key enabler. See figure 1 for a prototypical network topology.

In brief, the PXE protocol operates as follows. The client initiates the protocol by broadcasting a DHCPDISCOVER containing an extension that identifies the request as coming from a client that implements the PXE protocol. Assuming that a DHCP server or a Proxy DHCP server implementing this extended protocol is available, after several intermediate steps, the server sends the client a list of appropriate Boot Servers. The client then discovers a Boot Server of the type selected and receives the name of an executable file on the chosen Boot Server. The client detects which file transport protocol that the Boot Server support by interpreting PXE DHCP options. If the Boot Server supports FTP, then the client uses FTP to download the executable from the Boot Server. Otherwise, the client uses TFTP [23] to download the executable from the Boot server. Finally, the client initiates execution of the downloaded image. At this point, the client's state must meet certain requirements that provide a predictable execution environment for the image. Important aspects of this environment include the availability of certain areas of the client's main memory, and the availability of basic network I/O services.

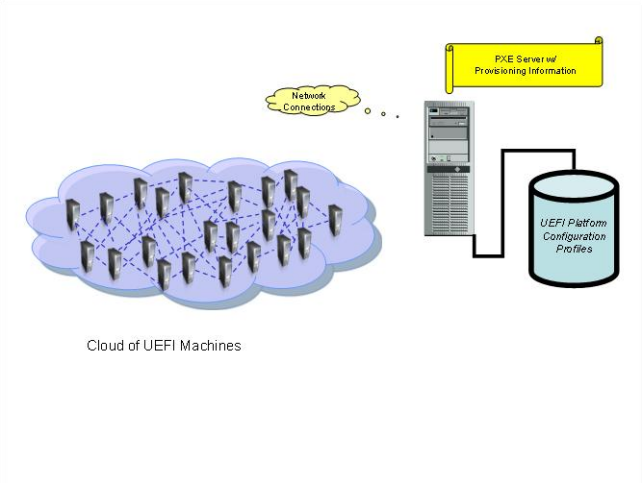


Figure 1 Cloud provisioning

The UEFI PXE extensions will augment the UDP-based TFTP downloads of PXE2.1 w/ the TCP-based FTP download. In addition, the extensions will support both IPV4 and IPV6. For IPV6, the platform should adhere to the IPV6 node requirements in RFC 4294. And as IPV6 requires IPSec, the platform firmware shall support RFC 4306.

An overview of the UEFI network stack is shown below:

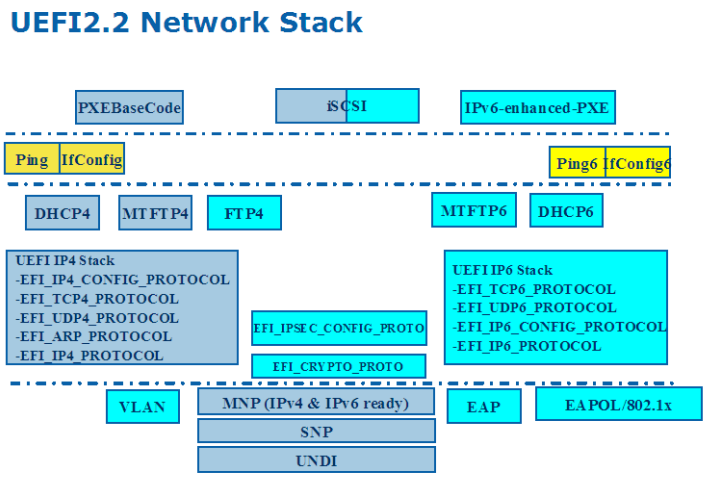


Figure 2 UEFI network stack

The notable aspect of this design is the 'dual stack' nature, namely there is both a IP4 and IP6 stack. This allows for interoperability in a network w/ both IP4 and IP6, IP4-only, or IP6-only. This is the typical

deployment model used by operating systems in supporting IP6 and maintaining IP4 compatibility.

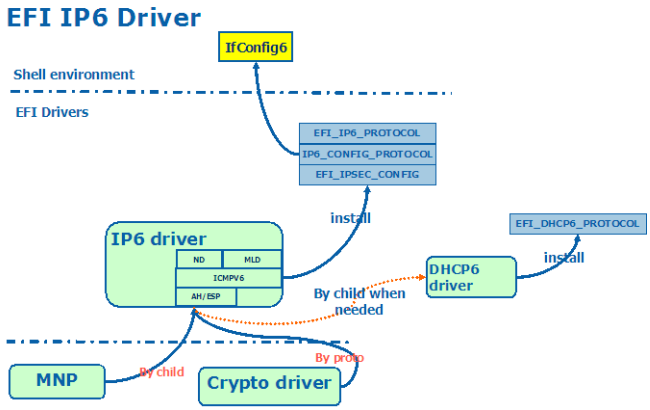


Figure 3 IP6 driver details

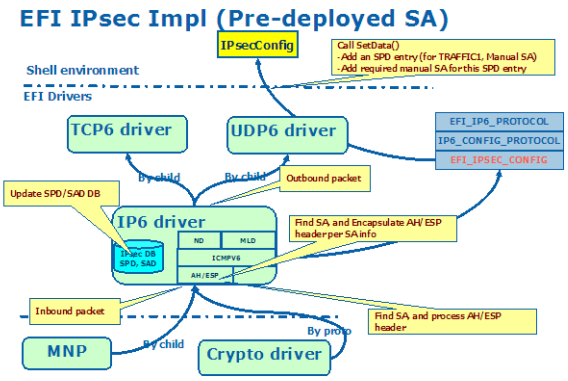


Figure 4 IPSec on the new network stack

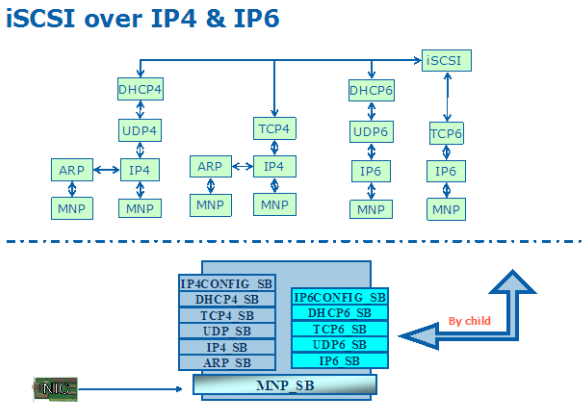


Figure 5 ISCSI support for IP4 and IP6

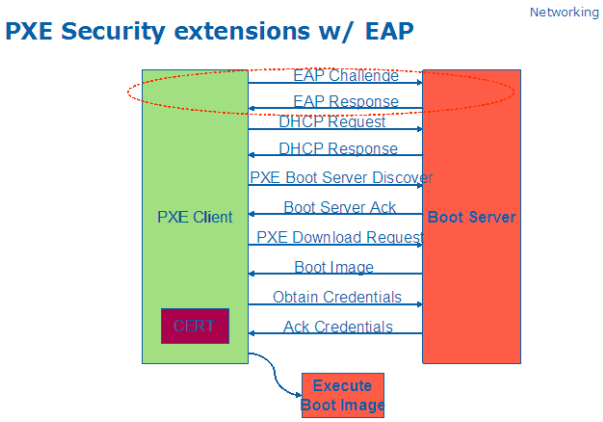


Figure 6 PXE boot w/ EAP/802.1x

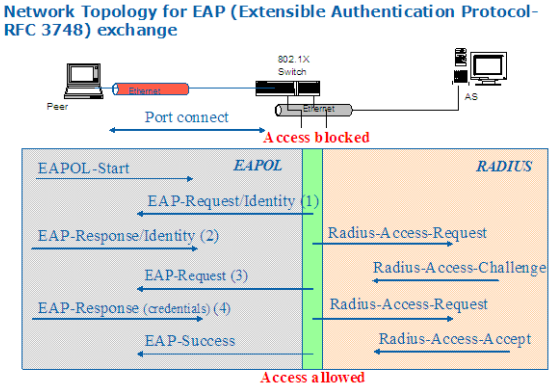


Figure 7 EAP exchange

In addition, the generalization of the transport and server name will also allow for booting via HTTP. This has the advantage of being a routable protocol and leveraging the plurality of web servers in the market. Since HTTP is based upon TCP, it should also be more robust that UDP-based TFTP. In addition, TLS in single or double-ended authentication mode will allow for various trust model deployments

The PXE Loadfile() capability serves as just one of many possible UEFI pre-OS applications. Other applications include but are not limited to iSCSI and PXE. One scenario therein includes: Mount a remote iSCSI LUN and create a UEFI simple system system instance, PXE boot an operating system installer program, and then install the operating system into the iSCSI-mounted filesystem instance. As such, the multi-application nature of the UEFI network stack will allow for these concurrent “PXE” and “ISCSI” operations.

In addition, the other UEFI networking capabilities are

composable with the above-listed flow. For example, an 802.1x [13] controlled switch can be used in wired scenarios where there are DHCP vulnerability concerns in order to authenticate the UEFI client before it joins the network. Or in wireless scenarios, the Extensible Authentication Protocol (EAP) [11] / 802.1x [13] support is an integral element of 802.11i support such that UEFI enables wireless PXE boot scenarios, too.

As discussed above, at some point in time prior to performing a secure network boot, it will be necessary to provide a shared secret to both the client that boots the authenticated boot image and the server from which the image is served. In one embodiment, the Extensible Authentication Protocol is employed as the means for communicating authentication information between a Supplicant (EAP peer) and an Authenticator (EAP server). EAP is a general protocol that supports multiple authentication mechanisms. For example, through the use of EAP, support for a number of authentication schemes may be added, including smart cards, Kerberos, Public Key Encryption, One Time Passwords, and others.

An exemplary EAP-based shared secret (i.e., key) exchange is shown in Figure 8. The process involves two phases, including a phase 1 under which authentication is performed, and a phase 2 during which secure messages are exchanged. The process begins at an EAP server (Authenticator), which sends an EAP request message containing a SessionID and a Sid to an EAP Peer (Supplicant). In one embodiment, the SessionID is a 256-bit random value generated by the authenticator. The Sid comprises and EAP server identity. In one embodiment, the Sid comprises a network access identifier as specified by RFC 2486 [8].

In response to receiving EAP request message, EAP peer returns an EAP response message including a concatenation of a first hash Hash1, a Pid value, a Kid value, a public key (PubKey) and a random number Prandom selected by the EAP peer. Hash1 is an SHA1 hash of the first message (i.e., EAP request message). Pid identifies the EAP Peer, and hence, the owner or device key. In one embodiment, Pid comprises a GUID (globally unique identifier). Kid identifies the symmetric key the EAP peer expects to use in this context. PubKey is EAP peer's public key portion of the device key.

The EAP server then sends an EAP request message containing a second hash Hash2 comprising a hash on EAP response message, and an SAuth value, which comprises a concatenation of 3 items, encrypted under EAP peer's public device key: (1) session key Ks, (2) value of Hash2 repeated, and (3) an HMAC-SHA1-96 digest on the concatenation of Ks | Hash2. In response, EAP peer returns an EAP response containing a third hash Hash3 comprising a hash on EAP request message, and a Mac3 value, comprising an HMAC-SHA1-96 digest of Hash3 under the KCK portion of Ks. This completes the phase 1 authentication process.

In one embodiment, Ks comprises a 60-octet (480-bit) key with an internal structure of three 20-octete (160-bit) subkeys, including:
KCK – 1st octet of Ks. Key confirmation key.
KDK – 2nd octet of Ks. Key derivation key.
KEK – 3rd octet of Ks. Key encryption key.
During phase 2, the secure messaging phase, messages are exchanged in a secure manner under which an encrypted format is used that references the previous message received at each participant. This is depicted by the "n" and "n + 1" nomenclature shown in Figure 9. Each message includes a concatenation of a hash on the previous message (e.g., Hash(n)), a one-byte ID value identifying the type of data being conveyed, Adata, identifying data that is authenticated, Edata, identifying encrypted data, and an Mac value, comprising an HMAC-SHA1-96 digest of the other fields in the message, using the KCK portion of the session key Ks. In one embodiment, the encrypted Edata only includes keys and keying material. During the phase 2 operations, a shared secret, such as a key, may be securely exchanged between the EAP peer and EAP server.

Figure 9 shows a set of message exchanges comprising an encapsulation of the authentication and secure boot image process involving a PXE (pre-boot execution environment) client, a DHCP (or a DHCP/Proxy) server, and a boot server. In addition to being hosted by separate machines (as shown), DHCP server and boot server may be co-located. DHCP server is also representative of an authentication server, in general.

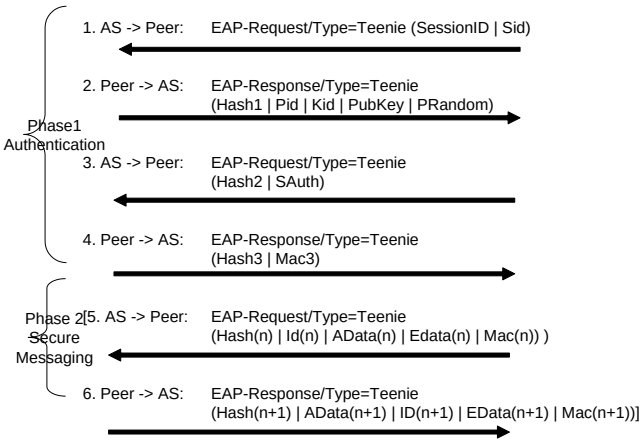


Figure 8 EAP authentication exchange

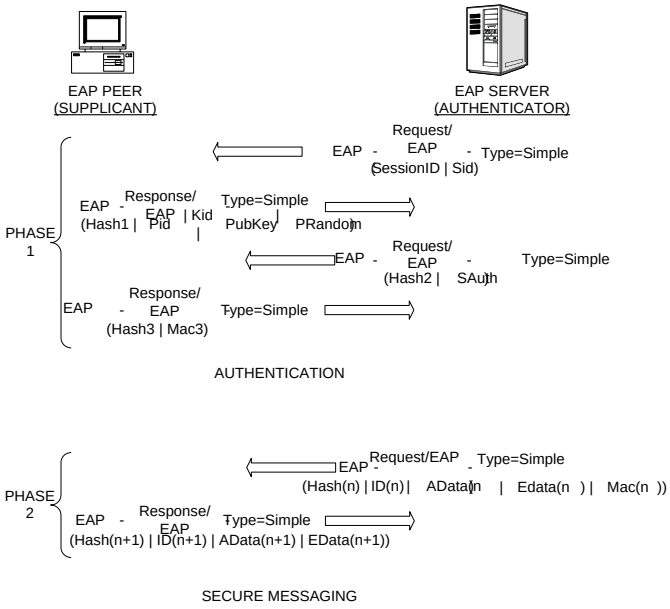
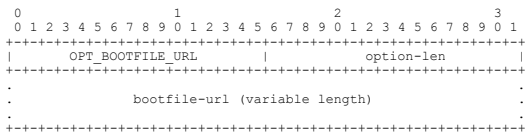


Figure 9 More details on EAP authentication

1.1.3 IPV6 extensions

In addition to having the ability to support richer downloads, UEFI also enables the ability to have a discoverable IPV6 network boot on PXE and ISCSI. This is accomplished via the definition of new option tags for DHCPv6 [7]. Even though IPV6 supports autoconfiguration in order to get an IP address, many network operators/IT prefer to manage their nodes with DHCP. As such, migrating today's PXE & ISCSI discovery to IPV6 is important to map today's IPV4 scenarios to IPV6.

In reviewing [7], though, the IETF work attempts to generalize the network boot process. This generalization takes the form of having the bootfile name as a URL, namely



This allows for booting from today's connectionless/UDP-based download such as PXE, connection-oriented ISCSI,FTP, or HTTP/HTTP-S. This choice of transport is expressed in the URL as 'tftp :', 'iscsi :', 'ftp :', 'http,' or 'http-s,' respectively. After the colon in the URL will be the path and filename of the network boot program, such as elilo.efi [2].

Security and pre-boot networking

Today, standards-based platforms are characterized by the Pre-boot Execution Environment (PXE) for pre-OS networking. PXE has two components: the pre-OS firmware API's to allow OS-absent networking in order to load an agent across the network, and the second is a collection of IETF RFC's for the wire protocols to use with the boot server. In the case of BIOS PXE and EFI1.10, the PXE includes DHCP, UDP, and TFTP. The BIOS version of the firmware API's are called the base code interfaces.

For EFI1.10, there is a set of C-callable API's for the base-code. That is further decomposed into a modular stack for UEFI2.0.

Beyond UEFI2.0, the UEFI Networking Subteam in UEFI2.2 added both IPV6 and IPSec support in the UEFI2.2 specification. This will allow for end-to-end security. In addition, layer 2 security in the form of the Extensible Authentication Protocol (EAP) is also being added to the network boot flow.

EAP admits many authentication methods, including but not limited to EAP-TLS, EAP-CHAP, and EAP-TNC [14] (the network access control protocol endorsed by the TCG). Finally, below shows a use of the TPM and EAP for enrolling a machine on the network.

The use of EAP is to address the hijack vulnerabilities of the DHCP protocol. To-date, use of authenticated DHCP or emergent proposals such as DHCP-EAP have been slow in emerging. As such, prefixing the DHCP request with a 802.1x/EAP request allows for encapsulating the DHCP negotiation and providing a degree of access control.

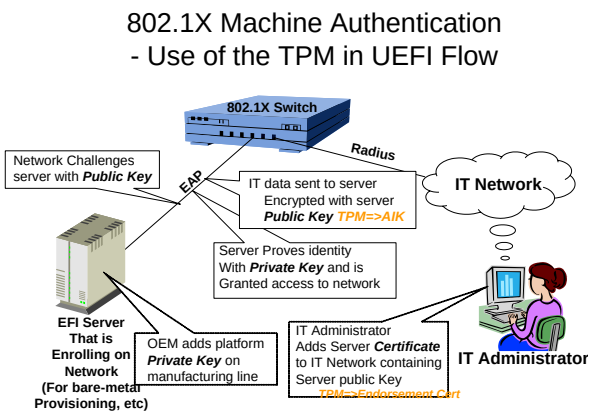


Figure 1-10 Enrollment across the network w/ a TPM

Other futures – AMT and UEFI synergy

Other proposed synergies include but are not limited to the Manageability Engine of Active Management Technology (AMT) [16] for storing the signature keys used during the UEFI 2.1 Secure Boot mentioned earlier.

This adds a capability for a platform owner to ensure that the firmware is only executed if in an owner-authorized fashion, such as signed components. Today the Trusted Platform Module (TPM) and the techniques promulgated by the Trusted Computing Group entail the use of the TPM as a Root-of-Trust for Storage (RTS) and Reporting (RTR) via the Platform Configuration Registers (PCR's) and Storage Root Key (SRK), respectively. The TPM is a passive piece of hardware, though. The platform firmware (or ucode or …) is the Root of Trust for Measurement (RTM). Recall that the UEFI Secure Boot adds a Root of Trust for Informant of Validation (RTE/RTV), though, which enables the "Secure Boot." This secure boot will entail the use of Authenticode-based signatures applied to the UEFI images [9] and will allow for detecting an integrity violation of the NBP in transit across the network, such as by a MITM (man-in-the-middle) attack.

Whereas RTM will record state and continue execution irrespective of the measurement, an RTE and "Secure Boot" will in fact stop the boot if the software state does not meet some integrity metric, such as a hash in a white-list or a digital signature.

UEFI enables both cases but advocates the latter since the list of possible hashes is an unbounded, management nightmare; public keys allow for a level of indirection to map the keys to a few trusted sources, thus easing the management problems associated with deployment. The problems being addressed with UEFI and AMT entail the difficulties faced with the management of credentials/public keys, location of the public key storage, inability to perform out-of-band secure firmware provisioning and updates.

This AMT-assist will also complement efforts in measured boot/platform security here at Intel, such as the recently made public EFI TCG Protocol and Platform Specifications. The problem with open platform is where to store the credentials (public key used for the signature key) and how to provision the keys and system (i.e., take ownership and get the keys into the system). This is where AMT and its network-facing, out-of-band capabilities comes into play.

This design uses the ability of the AMT to both provide a way to also check for public key/certificate revocation. Today there exist Certificate Revocation Lists (CRL's) that entail checking w/ the Certificate Authority. Today, most firmware lacks the type of sophisticated networking, as found in AMT and its Web Services stack, to perform this type of action. Revocation of the keys via AMT would be another leadership feature enabled by conjoining Server AMT and UEFI Secure Boot.

Other futures – NAC/AMT/UEFI boot

This art presents a generic mechanism for providing secure boot capabilities and in one instantiation speaks to covering or encapsulating today's PXE TFTP-based flow inside of a secure session. This art will leverage any tunnel able EAP (Extensible Authentication Protocol) method and TLV (Type-Length-Value) elements [12], to encapsulate a portion or all of the TFTP PXE flows. In this context, the term TLV is synonymous to AVP (attribute value pair), both defining a generic mechanism to encapsulate any arbitrary data.

This is akin to how Cisco and Network-Access Control (NAC) is used today for conveying device posture and returning an associated policy. Leveraging such a mechanism for PXE will preserve today's PXE server investment and client flows by augmenting them with an exterior secure transport.

Details of this design are described in the following sections. Additional claims around leveraging Intel® AMT providing some of the critical services in one manifest of this design provides an inherent Intel value add, where Intel® AMT may be leveraged to provide some of the security services, allowing existing client implementations of PXE to work virtually unmodified.

This provides a mechanism for hardening this exchange in one of the following ways.

In one manifestation of this design, the client (native host or Intel® AMT) provide the network access capabilities appropriate to the network access framework. In 802.1X networks, this is in the form of an 802.1X supplicant, executing an appropriate EAP method for authenticating the client to the Network Access Point (NAP), which may be a switch or an Access Point (AP). In non 802.1X networks, this manifests itself in the EAP protocol being conveyed over a UDP exchange (EAP-UDP). Furthermore, in remote access scenarios, this may be instantiated via a VPN connection, such as leveraging an EAP method over an Internet Key Exchange (IKE) version 2 [19] protocol for IPsec based VPNs. Leveraging any multitude of these protocols, which support device authentication via EAP allows initial device access to the network. Once a secure channel has been established between the client and the Authentication Server (AS), residing behind the NAP, a generic TLV / AVP mechanism may be used to enable security for PXE provisioning. Various forms of EAP-TLVs are widely used in the industry to convey different data objects. This design proposes leveraging a TLV to secure the PXE image in one of two ways.

The TLV exchange within EAP is used by the client to request data from the PXE server. The client provides a nonce value to the server to ensure liveliness of the exchange and mitigate any replay threats. The server returns one or more cryptographic keys (in native or wrapped form) to the client. This key (or some derivative) can be used to provide cryptographic security for transferring the PXE image from the server to the client. In this manifest, the client does not need to store any state associated with the key and is provided fresh a key by the back end server (PXE or AS or some other trusted entity in the network). If some other entity (e.g. consolidated policy server) is providing this key, then it is the domain administrator's responsibility to ensure that that key is also accessible to the PXE server. The mechanisms around key synchronization may be performed in a multitude of ways and is beyond the scope of this design. E.g. In a simpler embodiment of this design, the PXE server stores / generates / has direct access to this key. The key may be symmetric or asymmetric, based on the instantiation of the model. Leveraging the inner EAP TLV method to provide a key to the client ensures that the client and PXE server share a unique key (symmetric or asymmetric), which allows any further data exchange to be cryptographically secured from any intermediate, malicious adversaries. The selection of an appropriate cryptographic algorithm used to secure this data is beyond the scope of this design and is dependent on cryptographic capabilities between the client and the PXE server. NIST recommends leveraging AES [17] (Advanced Encryption System) for data encryption, but other methods may be equally applicable. The server leverages this key to protect the PXE image (integrity protected to ensure it cannot be modifier en-route, as well as encrypted (if desired) to prevent visibility into the image. The client uses this key to validate the integrity of this image before execution. One key benefit of this approach is that after the key is retrieved using the TLV method, the existing PXE protocol (TFTP) can be used natively to obtain the image in a secure manner, as the image itself is now cryptographically protected. The TLV channel may also be used to securely convey additional data, indicating the address of the PXE server to be contacted.

b) The second embodiment of this design allows the whole image to be downloaded via the TLVs. As the TLV mechanism is generic, it can be extended to carry a whole protocol within the TLV exchange. In this embodiment, once the outer EAP channel is established, the client wraps any further exchanges for PXE within the TLVs and executes the protocol to obtain the boot image. The outer EAP channel serves as a tunneling protocol for any further exchanges. On the client, this requires an interface to the EAP supplicant to inject / retrieve data. The TLV component of the supplicant is responsible of wrapping / unwrapping any data provided into TLVs and conveying this across the EAP tunnel. The server receiving this data similarly unwraps the TLV headers and forwards the data to the appropriate destination. If native PXE frames are sent onto the network from the EAP server, then the server needs to act as a proxy for the client to ensure the reverse data packets are routed back to the server for encapsulation in the EAP-TLV tunnel.

Related Work

There are several technologies for over-the-air (OTA) and on-the-wire provisioning. Most of these entail a vertically integrated provisioning and client stack. This design leverages UEFI infrastructure to have shrink-wrap provisioning server and node network boot experience.

Conclusion

This art provides a series of capabilities for the network boot that allows for scale, IPV6/IPV4 mixed network support, and security for bare metal scenarios. Given the use of platform firmware for both bare metal and virtualized (e.g., UEFI guest firmware) boot and configuration of disaggregated

environments like the Cloud, these capabilities are more imperative.

2 References

[1] Unified Extensible Firmware Interface Specification Version 2.2 http://www.uefi.org.

[2] Elilo http://elilo.sourceforge.net/

[3] Vincent Zimmer, Michael Rothman, Robert Hale, *Beyond BIOS: Implementing the Unified Extensible Firmware Interface Specification with Intel's Framework*. Intel Press, September 2006. ISBN 0-9743649-0-8.

[4] Vincent Zimmer, "Platform Trust Beyond BIOS Using the Unified Extensible Firmware Interface," in *Proceedings of the 2007 International Conference on Security And Manageability, SAM'07*, CSREA Press, June 2007

[5] Vincent Zimmer, "System Isolation Beyond BIOS Using the Unified Extensible Firmware Interface," in *Proceedings of the 2008 International Conference on Security And Manageability, SAM'08*, CSREA Press, June 2008

[6] UEFI Platform Initialization Specification, Version 1.1, Volumes 1 -5 www.uefi.org

[7] IPv6 net-boot extension
http://tools.ietf.org/html/ietf-dhc-dhcpv6-opt-netboot-03
http://www.ietf.org/proceedings/08nov/slides/dhc-0.pdf
http://www.ietf.org/proceedings/08nov/slides/dhc-1.pdf
http://www.ietf.org/internet-drafts/draft-zimmer-dhc-dhcpv6-remote-boot-options-01.txt
http://tools.ietf.org/html/draft-zimmer-dhc-dhcpv6-remote-boot-options-00

[8] Network Access Identifier RDC 2486
http://www.faqs.org/rfcs/rfc2486.html

[9] Authenticode
http://www.microsoft.com/whdc/winlogo/drvsign/Authenticode_PE.mspx

[10] IP Security, RFC 2401
http://www.ietf.org/rfc/rfc2401.txt

[11] Extensible Authentication Protocol (EAP), RFC 3748, http://www.rfc-archive.org/getrfc.php?rfc=3748

[12] NAC http://www.ietf.org/internet-drafts/draft-thomson-nacp-01.txt

[13] IEEE 802.1x Port-based access control
http://www.ieee802.org/1/pages/802.1x.html

[14] Trusted Network Connect Specifications
https://www.trustedcomputinggroup.org/specs/TNC/

[15] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica and Matei Zaharia, "A Berkeley View of Cloud Computing," EECS-2009-28, http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/

[16] Active Management Technology (AMT)
http://www.intel.com/technology/platform-technology/intel-amt/index.htm

[17] Advanced Encryption Standard (AES)
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[18] Preboot Execution Environment (PXE)
http://www.pix.net/software/pxeboot/archive/pxespec.pdf

[19] Internet Key Exchange version 2 (IKE)
http://tools.ietf.org/html/rfc4306

[20] EFI TCG Protocol
https://www.trustedcomputinggroup.org/specs/PCClient/TCG_EFI_Protocol_1_20_Final_.pdf

[21] DoD IPV6 Standard Profiles for IPV6 capable products http://www.moonv6.org/lists/att-0506/DISR_IPv6_Product_Profile_draft_v.6_29Dec05.doc

[22] Analysis and Recommendations on the Exhaustion of IPv4 Address Space
http://www.nic.ad.jp/en/research/IPv4exhaustion_trans-pub.pdf

[23] Trivial File Transfer Protocol (TFTP), RFC 1350
http://www.faqs.org/rfcs/rfc1350.html

[24] Domain Host Controller Protocol (DHCP)
http://www.ietf.org/rfc/rfc2131.txt (IPV4),
http://www.isi.edu/in-notes/rfc3315.txt (IPV6)

[25] Hyper Text Transfer Protocol (HTTP) RFC 2616
http://www.faqs.org/rfcs/rfc2616.html

[26] File Transfer Protocol (FTP), RFC 959
http://www.ietf.org/rfc/rfc959.txt

[27] TCP/IP, RFC 793
http://www.faqs.org/rfcs/rfc793.html