Carnegie Mellon University
Software Engineering Institute

SECURING UEFI: AN UNDERPINNING TECHNOLOGY FOR COMPUTING

Vijay Sarvepalli (SEI)

May 2023

Of the beginnings

Each time you boot up your computer, software called firmware wakes up your computer, and often it remains active in the background, silently supporting the functions of the operating system. Originally, this software, known as the Basic Input/Output System (BIOS), contained a small amount of code without much more responsibility than to ensure that the operating system started up properly. Over decades, this firmware has grown in capability, size, and complexity. Many functions that used to be implemented either directly in hardware or in the operating system are now increasingly implemented in this critical layer of software.

Most modern desktops and servers have firmware based on a standard known as the Unified Extensible Firmware Interface (UEFI), which replaces BIOS. A typical UEFI-based firmware is composed of software components from several suppliers, often including code from open-source projects, all knit together by an original equipment manufacturer (OEM), such as a laptop manufacturer. These software components are primarily written in low-level programming languages like C that facilitate direct access to the hardware and physical memory. These software components require high-privilege access to the central processing unit (CPU). The Chain of Trust model in the UEFI standard is designed to enable secure cryptographic verification of these components, establishing assurances that only trusted software is executed during the early boot cycle [Wilkins 2016]. Even after the boot cycle is complete, UEFI still provides an interface to the operating system to enable configuration changes or software updates to the firmware.

Unlike the operating system, UEFI software remains invisible to most of us, despite its critical role in the functioning of a modern system. Because of its criticality and invisibility, vulnerabilities in UEFI-related software pose high risks to system security. In an earlier blog, we explored why attackers find UEFI an attractive target for attacks [Sarvepalli 2022]. This paper highlights the technical and collaborative efforts to secure the UEFI-based firmware that serves as a foundational piece of modern computing environments.

Here be dragons

The UEFI specification defines the requirements for the technologies that enable a machine to boot securely and to initialize the low-level software required to support the operating system. The specification describes a framework for the development of hardware device drivers, a collection of application programming interfaces including those required to support power management, and additional capabilities such as SecureBoot. Together these features specify the interfaces that enable a fully operational UEFI firmware software stack. This section highlights some of the challenges for doing secure development using UEFI specification with currently available tools and frameworks.

Memory management in UEFI programming

UEFI software is designed to run in what is known as the System Management Mode (SMM), an isolated execution environment supported by x86 CPUs that executes with high privileges. This SMM operates on a dedicated and secure section of volatile System Management Random Access Memory [Wilkins 2015]. The SMM transition of the CPU is triggered by a System Management Interrupt (SMI) Handler [Yao 2019b]. An SMI Handler can be generated either from hardware (such as a device) or from software, should be handled with high priority, and cannot be ignored [Microsoft 2020]. The developer is expected to ensure that the System-Management Range Register is programmed to restrict access to the System Management Random Access Memory region and prevent any unauthorized access. The operating system should use the abstract Communications Buffer interface and should not directly access System Management Random Access Memory or directly control the execution of the SMM [Yao 2019c].

Because memory access and SMM execution is complex, it is easy for software developers to get it wrong, introducing security bugs [Boone 2023]. Attackers can exploit these vulnerabilities to modify the behavior of code running in a high-privileged mode of the CPU, for example, causing a device driver, such as a Driver eXecution Environment, to leak memory or triggering an EFI Variable change that subverts the normal boot sequence. Testing UEFI code presents a variety of challenges that are unique to UEFI firmware environments and makes applying modern testing tools (e.g., static application testing tools) more difficult.

Challenges in establishing a root of trust

UEFI Secure Boot is a feature defined in the UEFI Specification that guarantees that only valid third-party firmware code can run in the OEM firmware environment [Yao 2019d]. UEFI firmware root of trust is the authoritative cryptographic source that can be relied on to verify the trustworthiness of the software used to bootstrap a computer's software from the early stages of the boot process [Chamorro 2020]. The goal is to provide hardware-initiated (immutable) validation that the firmware component's code has been provided by a trusted source and has not been tampered with [Zimmer 2016]. At each step of the boot process, a series of verification steps establish a chain of trust, where at each step a proven trustworthy component in turn validates the next, creating a transitive trust [Yao 2019a].

The system works well in theory; however, putting it into practice has presented a number of challenges that have not been overcome. A vulnerability in any given component can undermine the entire chain of trust, leading to the execution of untrusted code at some point before the operating system is started. For example, a recent attack dubbed BlackLotus chains a two-staged attack to undermine the root of trust—one to put a vulnerable EFI bootloader (a small piece of operating system loading software) in place and the second to use the vulnerability to undermine the chain of trust [Microsoft 2023]. Thus it uses a signed piece of vulnerable software to bypass security features in UEFI such as SecureBoot. Another set of problems can arise from a stolen or a compromised cryptographic signing key, as seen in the recent compromise of source code and private keys in Micro-Star International firmware [Lakshmanan 2023]. Once a private key has been compromised, any system that is built with a component from the compromised supplier is also now at risk [Goodin 2023]. As all of the UEFI components on a single device operate with equally high privileges and have no inherent isolation from each other, there is no way to limit the damage from one such compromised piece of firmware.

The current specification of the chain of trust also lacks sufficient fidelity to establish that all the relevant parts of UEFI-related memory can be trusted [Microsoft 2022]. For example, it is possible to verify certain parts of the content in the protected memory such as UEFI driver code, but not the data that is identified as EFI Variable, essentially the runtime variable used to manage settings of the firmware.

The UEFI standard addresses the risks due to compromised components—due to either a software vulnerability or a supply-chain compromise—through the Forbidden Signature Database (DBX).

The forbidden DBX dilemma

The DBX, also known as the UEFI Revocation List, is used during the boot sequence to determine whether any revocations apply to a particular binary that is stored in the PCI flash. A revocation list entry describes components in one of three ways:

- an entry with the hash of a specific binary
- a Public Key Infrastructure certificate (X509 form)
- the hash of a certificate

The UEFI specification requires that a security error is raised and the denied component is not processed if it matches a deny-list entry in the DBX. Similarly, there is an allow list that does the opposite checks after the forbidden checks have completed. A component that is not explicitly present in the allow list will also not be processed. The UEFI Forum maintains a publicly available DBX for many known-bad binaries and compromised certificates at https://uefi.org/revocationlistfile.

This DBX feature has proven to be difficult to manage in practice. The structure of DBX is complex because the structure depends on what is being blocked or allowed. For example, Microsoft's Authenticode signatures use a method of calculating the length of the header of the binary that is distinct from the traditional hash-based signature [Jones 2014]. These complexities could lead to flaws in implementation that allow for a deny list to be ignored, introducing the security risk from a vulnerable component. Further, the software that updates the DBX performs no validation or verification to ensure that the intended component is successfully blocked. Any attempts to add such stolen Public Key

Infrastructure certificates from compromised suppliers could render unusable a number of computers with a component from that supplier. Similarly, blocking vulnerable components such as bootloaders from operating system suppliers, using Forbidden DBX, can cause interruption to many systems. This limits the UEFI Forum from adding entries to Forbidden DBX that can operationally impact many users and enterprises. Finally, there is a burden on the end user or the enterprise device management to ensure that a DBX update is applied and audited. Because the DBX is stored in the limited-size PCI flash storage with other operational UEFI firmware, the storage size is limited; vendors must constantly work with such constraints and have not many tradeoffs to manage them. Firmware suppliers currently do not have a way to dynamically manage this list and deny access to insecure, vulnerable, or deprecated components.

Supply chain confusion

Supply chain issues are getting a lot of attention from the security community, much of it focused on what has been called "dependency confusion." Firmware is a veritable monument to dependency confusion. A typical OEM laptop or desktop may have 50 or more Pre-EFI Initialization modules and hundreds of Driver eXecution Environment modules inside its firmware [Matrosov 2022]. During the process of compiling, configuring, and assembling components into the intended final binary form, many components can end up modified and unscrutinized. A UEFI firmware supply chain typically involves independent BIOS vendors that inherit code from CPU manufacturers (and other reference implementations such as TianoCore or Project Mu). The independent BIOS vendors in turn provide their source code or a binary to original device manufacturers, who share the code, sometimes modified, with the final OEM.

It is the nature of the UEFI that vendors at different levels of the supply chain provide code, libraries, and components to other vendors along the supply chain. Software can of course be easily copied, modified, and appended, especially when the original source code is easily available, making it difficult to track versions and address bugs. The UEFI firmware supply chain is quite complex, and the final OEM device manufacturers, like Dell or Lenovo, have typically contributed less than 10% of the code. Essentially these OEM's may not have immediate visibility of close to 90% of the UEFI components included in the devices they sell. Each of these suppliers is also part of the transitive chain of trust that is entirely based on the security of their Public Key Infrastructure. Even if an OEM adheres to the proper rules and trusts only specific suppliers, the risk of one such supplier's private key information being compromised can undermine all the devices that inherited such trust [Roy 2021]. Further, a vulnerability found in a UEFI module or a boot loader from anywhere in this supply chain can have impacts up and down the chain, causing a ripple effect that risks the final assembled device. Finally, very few OEM vendors are able to generate and provide anything that comes close to a comprehensive software bill of materials of their UEFI components, making impact assessment and auditing a feat to accomplish for enterprise system managers and end users.

Synthesis of challenges

Stepping back, we can see that although in principle the UEFI defines a modular framework for assembling a firmware image for distribution, it does not provide the sort of modularized architecture we've come to associate with today's operating systems, hypervisors, and other execution environments. There are several phases to the boot process where these modules are executed with an optional capability to interact with other components. To make things even worse, in order to support certain configuration management features, some of these modules are also expected to accept untrusted input from the user or network and process it along with the stored information such as EFI Variables and device configuration data, for example, Boot Configuration Data. A successful attack and compromise of one of these modules could provide unrestricted access that could compromise the entire platform.

Why would threat actors target UEFI?

Malware families such as LoJax and BlackLotus represent threats that have used UEFI as a tool to gain persistence on systems after initial compromise. Persistence is one reason threat actors might find UEFI an attractive target. In this section, we discuss three other reasons that actors are drawn to attack UEFI: its relative invisibility to defenders, the slow or inconsistent patching cadence due to UEFI suppliers' lack of product security incident response team maturity, and access and execution privileges that, if compromised, could be used to avoid detection by any defender processes in the operating system.

Defenders seek recovery, attackers seek persistence: exploiting UEFI for long-term persistence

As threats have evolved, many companies have begun to acknowledge the reality that attackers will eventually gain some sort of access. Due to a flaw in either the setup or in the software, attacks are bound to find their way into systems. The focus lately has been on denying persistence or long-term access to the attackers. Some simple interruptions such as reboots, change of credentials, or riddance of the initial attack vector are all being adopted and even acknowledged as part of the National Institute of Standards and Technology (NIST) SI-14 Non-Persistence System Integrity Controls (Special Publication 800-53, Revision 5).

This basically means attackers now have to pursue methods that will give them both stealth and persistence. Many of the attacks against UEFI provide an attractive target for both objectives. Because UEFI infections can survive reboots and even reinstallation of the operating system, UEFI firmware is a profitable target. UEFI itself has many of the essential components of an operating system, such as networking, allowing the attacker a long-term ability to maintain access and even update the malicious firmware modules to maintain access indefinitely. Recent attacks such as MoonBounce and BlackLotus have illustrated how malware can use UEFI for persistence. These malware families have so far focused on highly visible targets within UEFI images. Microsoft's blog post about BlackLotus

detection outlines several ways that an administrative user in the operating system can detect a BlackLotus infection, for example [Microsoft 2023]. However, other (harder to access) features of the UEFI standard would provide attackers with much more invisibility.

The hidden enemy: you can't defend what you can't see

As the old adage goes, "absence of evidence is not evidence of absence." UEFI firmware has a huge security weakness. It's difficult to know what is actually present in a firmware at installation time and hard to perform runtime security monitoring [Rohner 2021]. Many vendors attempt to make it difficult to access and audit the PCI flash, where a persistent copy of the firmware is installed. There are few antivirus, anti-malware, and endpoint detection and response tools that can audit the UEFI boot sections, alert organizations on changes to the firmware, or attempt to recover from an infection. While the operating system itself can access UEFI through system and kernel calls to the application programming interface that UEFI specifically exposes and provides, it is difficult to guarantee a full reset of the SPI flash such that an infection has been entirely removed and the threat neutralized. The CPU itself limits this access, which further limits the ability of an operating system or application running under normal conditions to perform any activity to prevent malicious activity against UEFI firmware.

Patch me if you can: patching isn't the routine activity it should be

Patch Tuesday is the one entry you can expect to see on just about every IT and security professional's calendar. Every week security professionals learn about the latest issues and apply hot fixes, patches, and security updates to a number of systems that they manage. A lot of effort and time are spent planning, scheduling, and applying updates in a way that minimizes disruptions: these teams dread applying updates that require reboots, machine-level access, and physical access. While there are many good techniques to update UEFI firmware, they are difficult to implement in practice. There is always a risk of SPI flash corruption and potentially "bricking" the entire device.

UEFI specification allows for firmware updates to be supplied as UEFI Capsule updates using Firmware Management Protocol, which requires the kernel or Ring 0 privilege for installation. The UEFI Capsule has a well-defined structure with a Globally Unique Identifier and an update component that provides a reliable way to deliver updates. The producer of the system firmware is expected to package updates in a particular way, adding digital signatures that can be verified and using application programming interfaces to install them. A typical laptop or desktop may have 50 Pre-EFI Initialization modules and 180 Driver eXecution Environment modules inside its firmware. Depending on which firmware component is being updated, an essential security update may have to traverse a tortuous path through the supply chain of chip vendors, independent BIOS vendors, OEMs, original device manufacturers, and independent hardware vendors to appear in an end user's system. Sometimes these vendors also provide their own custom software to update and patch their components, independently of the final OEM or other supply-chain stakeholders. These updates may require manual intervention such as download, verification, installation, and, in some cases, cumbersome reset of the core components that require reboot (for example, the Trusted Platform Module). It should be assumed that the mean time to patch for firmware is potentially much longer than the 150 days observed in traditional

software patching [de Vries 2020]. Attackers can take advantage of these delays to target their attack during this vulnerable period and gain high-privilege access.

The M-mode phenomenon: even more privileged access

The RISC-V, an open-standard instruction set architecture, defines three modes of privilege levels. In increasing order of privilege, they are user-mode (U-mode), supervisor mode (S-mode), and machine mode (M-mode). In the M-mode is the highest privilege mode provided to firmware and boot loaders. The SMM available in x86 architectures is the most salient example of an implementation mode intended for similar high-privileged tasks. This mode of operation is below the operating system, which runs in Ring 0, and is typically identified as Ring -2 (pronounced ring minus two). This high-privileged mode is a very attractive target for attackers because it gives them a higher privilege than the operating system on the machine, while they benefit from the lack of tools to supervise and audit actions of processes running below Ring 0. Although the effort required to exploit UEFI firmware is high, the attacker who successfully exploits any weakness is rewarded with a persistent high-privileged access.

The road ahead: secure by design to modernize UEFI firmware

A staggering 286.2 million Windows-based PCs were sold in 2022, despite the 16.2% fall in PC sales that year. The Compatibility Support Mode that allows legacy BIOS to run on these devices is absent from almost all of them, virtually forcing the adoption of UEFI. UEFI is now actively running on hundreds of millions of machines worldwide. Each of these devices has firmware running on processors on the mother board and on the integrated (or latter-attached) peripherals. Fortunately, there are a variety of efforts to identify the actions required to secure the UEFI firmware ecosystem. Probably the most definitive source for guidance on UEFI can be found in the NIST *Platform Firmware Resiliency Guidelines* (SP 800-193) [Regenscheid 2018]. While it is difficult to predict the next threat and the goals of the adversary, if UEFI ecosystem partners fix the “known unknowns” in the UEFI firmware, they will make it harder for attackers to gain privileged footholds in these systems.

Build a robust attestation ecosystem

The discovery of the BlackLotus malware has highlighted how signed but vulnerable firmware is likely to remain a threat for a long time. The DBX represents only a part of the solution. An enhanced capability to dissect and block software that should no longer be trusted is essential. Dynamic Root of Trust for Measurement is one such approach for a dynamically verifiable trust that can restrict access to untrusted or newly introduced components [Brannock 2021]. Remote attestation is another capability that provides a trusted network resource that can dynamically verify code and add it to a trusted software list, either on first use or as needed. Remote attestation is already used by Apple’s iBoot and Google’s CoreBoot to verify any changes to firmware [Hudson 2019]. This type of attestation, with a

sealed secret and a monotonic firmware versioning, can prevent tampering, rollback attacks, and any attempts to use compromised components to reduce the integrity of a system [Rutkowska 2015]. The next version of this type of technology, such as SecureBoot, should inherently be able to provide attestation capabilities to ensure a dynamic chain of trust. Mechanisms to block or restrict the execution of a group of binaries and disabling sets of vulnerable components should also be made inherently available as part of the firmware verification.

Improve the memory safety of critical UEFI code

UEFI firmware security requires rigorous discipline in memory management. The reference implementations, such as Tianocore and Project Mu, should provide this discipline by default. Vendors are likely to reuse the widely available code to develop their implementations. These reference implementations have not been adequately verified.

Memory management verification can also enhance the adoption of capabilities. For example, an input/output memory management unit protects against direct memory access as the default case at the early boot phases [Yao 2017]. Vendors and their security partners should thoroughly audit and restrict the use of critical SMI Handlers and use SMI Handler Profiles to ensure limited and secure use of high-privilege access to only necessary components. SMI Handlers should be avoided when not needed, as recommended on the UEFI Forum [Wilkins 2015]. Organizations should develop software auditing capabilities, such as CodeQL, to support developers by profiling SMM code development so that the memory-safe programming audit itself is codified and enforceable. The UEFI community should support, encourage, and reward researchers, educational institutions, and partners to contribute to such enhancements and pursue such memory-safe methods for the industry.

Apply least privilege and component isolation principles to UEFI code

As noted, the design of a UEFI image is in some sense both modular and flat. As UEFI looks more and more like its own operating system, today's enclaving technologies—such as Intel® Software Guard Extensions, Advanced Micro Devices® Secure Encrypted Virtualization, and the Arm® TrustZone®—need to be extended into UEFI. There is also a need for memory isolation between boot applications. In November 2022, Microsoft announced an aspiration toward achieving this goal using EFI_MEMORY_ATTRIBUTE_PROTOCOL, but much of the current code continues to work without any such isolation. At this time, most firmware implementations have not adopted this isolation model. In 2023 UEFI PlugFest also brought to life the Universal Scalable Firmware efforts, which include a specific call to action to deprivilege and thereby isolate the UEFI Third-Party Option ROM (read-only memory) using Ring 3 OEM SMM. This type of principle of least privilege is also much needed to reduce risk that attackers could gain untethered access to the CPU from UEFI modules or components. As UEFI firmware continues to grow with multiple firmware components, at times dynamically introduced, component isolation has become an essential design need in the UEFI ecosystem. The design of the UEFI layer should incorporate mechanisms so that the consequences of the failure or compromise of any particular component do not introduce failure or compromise to the entire device.

Embrace firmware component transparency and verification

A software bill of materials (SBOM) is a nested inventory, a list of ingredients that make up software components. UEFI firmware includes many binary components built by a host of vendors, further adopted and modified as needed, with the OEM's contribution of code representing less than 10% of the total code. This code includes third-party libraries as well as code copied directly from third parties. In an ideal world, this SBOM is stored alongside the firmware image in the PCI flash and is updated routinely to reflect the most accurate picture of the contents of the image. In principle, most of the required capabilities are in place to maintain an accurate SBOM, yet few OEM vendors today are able to provide an SBOM that accurately represents the current firmware present on the system. The value of maintaining SBOMs in vulnerability management is undisputed, and accurate SBOMs are arguably one of the most important elements of supply-chain accountability. Integrating SBOMs into UEFI software development and firmware creation in the reference implementation can simplify efforts and bring cumulative value to accountability in the UEFI community.

Develop robust and non-intrusive patching

In an ideal future state, UEFI firmware patches will be applied routinely as part of normal IT operations. There are two challenges to realizing this ideal. First, the patching process must be standardized, and the mystery taken out of decisions to apply patches, so that organizations and end users have a straightforward way to apply patches. Second, this mechanism must be implemented in such a way that updates can be applied without interruption to operations. UEFI component vendors should adopt UEFI's standards-based transparent interfaces and avoid any custom software that could delay or disrupt automated and reliable device management [Rothman 2022]. There has been a gradual adoption of Linux Vendor Firmware Service [Richardson 2020] and Windows Update as standard ways of updating UEFI firmware, and this practice should be adopted and shared within the UEFI community. Vendors are also urged to explore uninterrupted and non-intrusive ways to apply UEFI firmware patches. Traditionally these patches require reboot, device reset, and, in some cases, an interactive user intervention or physical presence at the device. Vendors should provide updates that can support a wide variety of use cases from laptops, desktops, and servers to headless devices that support UEFI firmware.

The future of UEFI security

As alluded to earlier, the UEFI standard is here to stay and is only expected to grow in its usage and adoption. It is important for the UEFI community to embrace these challenges and respond to them in a timely manner. While we do not know how the threat landscape will evolve, we know about the gaps and threat motivators that have briefly been highlighted here. Our focused, consistent, and regular actions in the coming years should steer UEFI firmware more confidently into the unchartered territories into which our computing industry will keep forging ahead.

References and further reading

[Boone 2023]
Boone, J. *Stepping Insyde System Management Mode*. NCC Group. April 2023. https://research.nccgroup.com/2023/04/11/stepping-insyde-system-management-mode/

[Brannock 2021]
Brannock, K. & Ueno, A. *Intel® Hardware Shield: Trustworthy SMM on the Intel vPro® Platform*. Intel. May 2021. https://cdrdv2-public.intel.com/756963/DRTM-based-computing_whitepaper_FINAL_MAY2021.pdf

[Chamorro 2020]
Chamorro, D. & Chow, R. Anchoring Trust: A Hardware Secure Boot Story. *The Cloudflare Blog*. November 2020. https://blog.cloudflare.com/anchoring-trust-a-hardware-secure-boot-story/

[de Vries 2020]
de Vries, R. & Wennekers, J. How Do You Measure the Success of Your Patch Management Efforts? *Security Intelligence*. January 2020. https://securityintelligence.com/posts/how-do-you-measure-the-success-of-your-patch-management-efforts/

[Garrett 2015]
Garrett, M. Beyond Anti Evil Maid: Protecting Hardware from Early Boot Attacks. CoreOS. December 2015. https://lab.dsst.io/32c3-slides/slides/7343.pdf

[Goodin 2023]
Goodin, D. Leak of MSI UEFI Signing Keys Stokes Fears of "Doomsday" Supply Chain Attack. *Ars Technica*. May 2023. https://arstechnica.com/information-technology/2023/05/leak-of-msi-uefi-signing-keys-stokes-concerns-of-doomsday-supply-chain-attack/

[Hudson 2019]
Hudson, T. *Trust, Lies and Attestation*. Lower Layer Labs. 2019. https://hardwear.io/netherlands-2019/presentation/roots-of-trust-attestation-keynote-talk-hardwear-io-nl-2019.pdf

[Jones 2014]
Jones, P. The UEFI Security Databases. *The Uncooperative Organization*. October 2014. https://blog.uncooperative.org/uefi/linux/secure%20boot/2014/10/23/uefi-security-databases.html

[Lakshmanan 2023]
Lakshmanan, R. MSI Data Breach: Private Code Signing Keys Leaked on the Dark Web. *The Hacker News*. May 2023. https://thehackernews.com/2023/05/msi-data-breach-private-code-signing.html

[Matrosov 2022]
Matrosov, A.; Vasilenko, Y.; Ermolov, A.; & Thomas, S. Breaking Firmware Trust from Pre-EFI: Exploiting Early Boot Phases. Blackhat USA 2022. May 2022. https://i.blackhat.com/USA-

22/Wednesday/US-22-Matrosov-Breaking-Firmware-Trust-From-Pre-EFI.pdf (media: https://www.youtube.com/watch?v=Z81s7UIiwmI)

[Microsoft 2020]
System Management Mode Deep Dive: How SMM Isolation Hardens the Platform. *Microsoft Security Blog*. November 2020. https://www.microsoft.com/en-us/security/blog/2020/11/12/system-management-mode-deep-dive-how-smm-isolation-hardens-the-platform/

[Microsoft 2022]
UEFI Memory Mitigations. *Microsoft Build*. December 2022. https://learn.microsoft.com/en-us/windows-hardware/drivers/bringup/uefi-ca-memory-mitigation-requirements

[Microsoft 2023]
Guidance for Investigating Attacks Using CVE-2022-21894: The BlackLotus Campaign. *Microsoft Security Blog*. April 11, 2023. https://www.microsoft.com/en-us/security/blog/2023/04/11/guidance-for-investigating-attacks-using-cve-2022-21894-the-blacklotus-campaign/

[Regenscheid 2018]
Regenscheid, A. *Platform Firmware Resiliency Guidelines*. NIST SP 800-93. National Institute of Standards and Technology. May 2018. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-193.pdf

[Richardson 2020]
Richardson, B.; Zimmer, V.; Kinney, M. & Hughes, R. Capsule Updates & LVFS: Improving System Firmware Updates. FOSDEM 2020. February 2020. https://archive.fosdem.org/2020/schedule/event/firmware_culisfu/attachments/slides/3709/export/events/attachments/firmware_culisfu/slides/3709/FOSDEM_2020_Intel_Capsule_Update.pdf

[Rohner 2021]
Rohner, B. & Ruoff, T. DHS CISA Strategy to Fix Vulnerabilities Below the OS Among Worst Offenders. RSA Conference 2021. May 2021. https://static.rainfocus.com/rsac/us21/sess/1602603692582001zuMc/finalwebsite/2021_US21_TECH-W13_01_DHS-CISA-Strategy-to-Fix-Vulnerabilities-Below-the-OS-Among-Worst-Offenders_1620749389851001CH5E.pdf

[Rothman 2022]
Rothman, M. & Zimmer, V. Understanding UEFI Firmware Update and Its Vital Role in Keeping Computing Systems Secure. *Embedded Computing Design*. June 2022. https://embeddedcomputing.com/technology/security/software-security/understanding-uefi-firmware-update-and-its-vital-role-in-keeping-computing-systems-secure

[Roy 2021]
Roy, J. B. Understanding Windows 10 UEFI Secure Boot: How It Helps to Secure Pre-Boot Phase. *How to Manage Devices*. June 2021. https://www.anoopcnair.com/windows-10-uefi-secure-boot-guide/

[Rutkowska 2015]
Rutkowska, J. Towards Reasonably Trustworthy Laptops. UEFI Plugfest. December 2015. https://lab.dsst.io/32c3-slides/slides/7352.pdf

[Sarvepalli 2022]
Sarvepalli, V. UEFI – Terra Firma for Attackers. *SEI Insights*. August 2022. https://insights.sei.cmu.edu/blog/uefi-terra-firma-for-attackers/

[Schwartz 2022]
Schwartz, M. *Understanding and Preventing Dependency Confusion Attacks*. FOSSA. June 2022. https://fossa.com/blog/dependency-confusion-understanding-preventing-attacks/

[Wilkins 2015]
Wilkins, D. UEFI Firmware: Securing SMM. UEFI Plugfest. May 2015. https://uefi.org/sites/default/files/resources/UEFI_Plugfest_May_2015%20Firmware%20-%20Securing%20SMM.pdf

[Wilkins 2016]
Wilkins, R. & Nixon, T. *The Chain of Trust: Keeping Computing Systems More Secure*. UEFI. June 2016. https://uefi.org/sites/default/files/resources/UEFI%20Forum%20White%20Paper%20-%20Chain%20of%20Trust%20Introduction_Final.pdf

[Yao 2017]
Yao, J.; Zimmer, V. J.; & Zeng, S. *A Tour Beyond BIO: Using IOMMU for DMA Protection in UEFI Firmware*. Intel 2017. https://www.intel.com/content/dam/develop/external/us/en/documents/intel-whitepaper-using-iommu-for-dma-protection-in-uefi.pdf

[Yao 2019a]
Yao, J. & Zimmer, V. J. Intel® Boot Guard. *Understanding UEFI Secure Boot Chain*. June 2019. https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/secure_boot_chain_in_uefi/intel_boot_guard

[Yao 2019b]
Yao, J. & Zimmer, V. J. Memory Protection in SMM. *Understanding UEFI Secure Boot Chain*. June 2019. https://edk2-docs.gitbook.io/a-tour-beyond-bios-memory-protection-in-uefi-bios/memory-protection-in-smm

[Yao 2019c]
Yao, J. & Zimmer, V. J. Memory Protection in UEFI. *Understanding UEFI Secure Boot Chain*. June 2019. https://edk2-docs.gitbook.io/a-tour-beyond-bios-memory-protection-in-uefi-bios/memory-protection-in-uefi

[Yao 2019d]
Yao, J. & Zimmer, V. J. UEFI SecureBoot: Glossary. *Understanding UEFI Secure Boot Chain*. June 2019. https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/glossary

[Zimmer 2016]
Zimmer, V. & Krau, M. *Establishing the Root of Trust*. UEFI. August 2016. https://uefi.org/sites/default/files/resources/UEFI%20RoT%20white%20paper_Final%208%208%2016%20%28003%29.pdf

Legal Markings

This material is based upon work funded and supported by the Department of Homeland Security under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

DM23-0545

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu
Email: info@sei.cmu.edu