UEFI Firmware
Security Concerns and Best Practices

Fall 2017 UEFI Plugfest
October 30 – November 3, 2017
Presented by Dick Wilkins, PhD & Jim Mortensen
(Phoenix Technologies, Ltd.)
Copyright © 2017 Phoenix Technologies Ltd. All rights reserved.

PHOENIX TECHNOLOGIES LTD. MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION HEREIN DESCRIBED AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE OR NON-INFRINGEMENT. FURTHER, PHOENIX TECHNOLOGIES LTD. RESERVES THE RIGHT TO REVISE THIS DOCUMENTATION AND TO MAKE CHANGES FROM TIME TO TIME IN THE CONTENT WITHOUT OBLIGATION OF PHOENIX TECHNOLOGIES LTD. TO NOTIFY ANY PERSON OF SUCH REVISIONS OR CHANGES.
Contents

• Introduction
• Threats and Mitigation Guidelines
• Additional Concerns
• Validation Guidelines
• Next Steps
• Questions
Introduction
This Content

• This content is an update of presentations at the 2014 and 2015 Spring UEFI plugfest events

• An earlier but more comprehensive version may be found at:
  http://www.uefi.org/learning_center/industryresources
Introduction

• UEFI firmware is now widely deployed and has become a target for hackers and security analysts/researchers
• Poor implementations affect the credibility of the UEFI “brand” and market perception of all implementations
• As with all software implementations, there are going to be faults - (Phoenix is not perfect, even if we want to be)
• Phoenix would like to share some of our best practices in the interest of raising the quality and security of all UEFI implementations
Introduction

Firmware is software, and is therefore vulnerable to the same threats that typically target software

- Maliciously crafted input
- Elevation of privilege
- Data tampering
- Unauthorized access to sensitive data
- Information disclosure
- Denial of Service
- Key Management
- Etc.
Introduction

Firmware-Specific Threats

- Maliciously crafted input – Buffer overflows to inject malware
- Elevation of privilege – SMM code injection
- Data tampering – Modifying UEFI variables (SecureBoot, Configuration, etc.)
- Unauthorized access to sensitive data – Disclosure of SMRAM contents
- Information disclosure – SMM rooted malware; “secrets” left in memory
- Denial of Service – SPI flash corruption to “brick” the system
- Key Management – Private Key Management for signed capsule updates
We Are All At Risk!

Disclosures regarding UEFI BIOS security vulnerabilities look bad for the whole UEFI community!

So how do we protect against UEFI Firmware attacks?
Threats and Mitigation Guidelines
Threats and Mitigation Guidelines

Many organizations have provided disclosures of known issues and guidelines for developing more secure firmware.

Examples come from Intel, Microsoft, Mitre, NIST, Linux distros and others. Some are public and some are available only under NDA via direct communications with the involved companies.
Threats and Mitigation Guidelines

Key areas for concern:
• Firmware Flash Regions
• UEFI Variables in Flash
• Capsule Updates
• SMM
• Secure Boot
• Option ROMs
Threats and Mitigation Guidelines

Key areas for concern

• Firmware Flash Regions
• UEFI Variables in Flash
• Capsule Updates
• SMM
• Secure Boot
• Option ROMs
Threats and Mitigation Guidelines

Malware injected into the **address space** is transient, and will be cleaned up on the next boot.

Malware injected into the **firmware flash regions** is persistent and will run on every subsequent boot.
Threats and Mitigation Guidelines

SPI Flash Exploit

- All PEIMs in flash are mapped to the address space as a part of FV_Recovery
- An attacker with write-access to flash can inject malware into the firmware
- Malicious PEIMs can disallow flash updates, or cause destructive behavior (e.g., ‘brick’ the system)
- Malicious DXE drivers can disable security settings and install malicious code into the OS
- Malware in flash is persistent, and survives OS reinstall and hard drive reformat
Threats and Mitigation Guidelines

- All flash Lock bits must be appropriately set prior to running any untrusted code
- If flash writes are protected via SMI handlers, all SMM protection bits must also be appropriately set
- All Protected Range registers that block writes to flash address space must also be appropriately set and locked
On resume from S3:

- All flash Lock bits must be appropriately set prior to running any untrusted code

- If flash writes are protected via SMI handlers, all SMM protection bits must also be appropriately set

- All Protected Range registers that block writes to flash address space must also be appropriately set and locked
Threats and Mitigation Guidelines

On resume from S3:
- Scripts that re-initialize the platform must be secured against malicious modifications.
Threats and Mitigation Guidelines

Key areas for concern

• Firmware Flash Regions
• UEFI Variables in Flash
• Capsule Updates
• SMM
• Secure Boot
• Option ROMs
Threats and Mitigation Guidelines

- Ensure that all patches have been applied to Variable Services drivers
- Review custom implementations for similar vulnerabilities that have been patched in the core implementation
Threats and Mitigation Guidelines

- Lock Authenticated Variable regions as early as possible
- Separate integral configuration and security-based variables from those expected to be modified at runtime
- Reduce permissions to only what is needed
  - Remove RT access for POST-time variables
  - Set variables as Read-Only if they are not intended to be modified at runtime

Vulnerability Note VU#758382
Unauthorized modification of UEFI variables in UEFI systems

Original Release date: 09 Jun 2014 | Last revised: 03 Feb 2015

Overview
Certain firmware implementations may not correctly protect and validate information contained in certain UEFI variables. Exploitation of such vulnerabilities could potentially lead to bypass of security features and/or denial of service for the platform.

Description
As discussed in recent conference publications (CanSecWest 2014, Syscan 2014, and Hack-in-the-Box 2014) certain UEFI implementations do not correctly protect and validate information contained in the 'Setup' UEFI variable. On some systems, this variable can be overwritten using operating system APIs. Exploitation of this vulnerability could potentially lead to bypass of security features, such as secure boot, and/or denial of service for the platform. Please refer to the conference publications for further details.

Impact
A local attacker that obtains administrator access to the operating system may be able to modify UEFI variables. Exploitation of such vulnerabilities could potentially lead to bypass of security features and/or denial of service for the platform.
Threats and Mitigation Guidelines

Key areas for concern

- Firmware Flash Regions
- UEFI Variables in Flash
- Capsule Updates
- SMM
- Secure Boot
- Option ROMs
Secure Capsule Updates rely on proper signing, private key management, validation, and rollback protection

- NIST SP 800-107 provides guidelines for hash algorithm usage
- NIST SP 800-57 provides guidelines for key management
- NIST SP 800-147(b) provides guidelines for secure BIOS Updates
- NIST SP 800-193* provides general firmware resiliency guidelines, including firmware update mechanisms

* draft May 2017
Threats and Mitigation Guidelines

- Ensure that all patches have been applied to Capsule Update drivers
- Review custom implementations for similar vulnerabilities that have been patched in the core implementation
- Enforce Signed Capsule Updates
- Enforce Rollback Protection
- Use an HSM or Signing Authority for private key protection
Threats and Mitigation Guidelines

Key areas for concern

• Firmware Flash Regions
• UEFI Variables in Flash
• Capsule Updates
• SMM
• Secure Boot
• Option ROMs
What is SMM?

• Highly privileged processor mode
• Entered through a System Management Interrupt (SMI)
• Processor saves its context, services the SMI, then restores context and resumes
• SMM code has full visibility of all address space and devices
• Transition is transparent to the rest of the system
Threats and Mitigation Guidelines

SMM Mode

- SMM code has full access to all system memory and devices
- SMM code is not bound by OS Kernel or Hypervisor protections
- SMM code can read all of memory, modify memory contents, and even overwrite critical system files and data on storage mediums
Threats and Mitigation Guidelines

SMM Mode Exploits

- During an SMI, all code runs with SMM-level privileges (Ring -2) regardless of where it resides.
- Malware resident in SMRAM has full access to all system memory and devices.
- Legitimate code in unprotected memory can be modified by Ring 0 malware.
- Modified code called by an SMI handler runs with SMM-level privileges (Ring -2) and gains full access to the system.
Threats and Mitigation Guidelines

- SMM code must never call code outside of SMRAM because an attacker could have maliciously modified that code.
- SMM code must validate input parameters from untrusted sources to prevent buffer reads/writes that extend into SMRAM.
- SMM code must copy input parameters and validate and use the copy, to prevent time-of-check-time-of-use (TOCTOU) vulnerabilities.
Enable Hardware Protections

- Lock SMRAM as early as possible
- Lock SMI control registers
- Enable hardware NX protections for addresses outside of SMRAM (if supported)
- Enable paging NX protections for addresses outside of SMRAM
Threats and Mitigation Guidelines

Key areas for concern

• Firmware Flash Regions
• UEFI Variables in Flash
• Capsule Updates
• SMM
• Secure Boot
• Option ROMs
Threats and Mitigation Guidelines

- UEFI Variables that contain Secure Boot settings must be locked and protected from unauthorized modification.

Vulnerability Note VU#758382
Unauthorized modification of UEFI variables in UEFI systems

Overview
Certain firmware implementations may not correctly protect and validate information contained in certain UEFI variables. Exploitation of such vulnerabilities could potentially lead to bypass of security features and/or denial of service for the platform.

Description
As discussed in recent conference publications (CanSecWest 2014, Syoscan 2014, and Hack-in-the-Box 2014) certain UEFI implementations do not correctly protect and validate information contained in the 'Setup' UEFI variable. On some systems, this variable can be overwritten using operating system APIs. Exploitation of this vulnerability could potentially lead to bypass of security features, such as secure boot, and/or denial of service for the platform. Please refer to the conference publications for further details.

Impact
A local attacker that obtains administrator access to the operating system may be able to modify UEFI variables. Exploitation of such vulnerabilities could potentially lead to bypass of security features and/or denial of service for the platform.
Threats and Mitigation Guidelines

- SMM code must never call code outside of SMRAM as this could allow bypass of Secure Boot protections.
Threats and Mitigation Guidelines

- All flash Lock bits, SMM protections, and Protected Range registers must be properly set to prevent bypass of Secure Boot protections.

Vulnerability Note VU#766164
Intel BIOS locking mechanism contains race condition that enables write protection bypass

Overview
A race condition exists in Intel chipsets that rely solely on the BIOS_CNTL/IOSWGE and BIOS_CNTL/OSWGE bits as a BIOS write locking mechanism. Successful exploitation of this vulnerability may result in a bypass of this locking mechanism.

Description
CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization (Race Condition)
A race condition exists in Intel chipsets that rely solely on the BIOS_CNTL/IOSWGE and BIOS_CNTL/OSWGE bits as a BIOS write locking mechanism. According to Corey Kallenberg of The MITRE Corporation:

"When the BIOS_CNTL/IOSWGE bit is set to 1, the BIOS is made writeable. Also contained with the BIOS_CNTL/OSWGE register is the BIOS_CNTL/OSWGE bit (BIOS Lock Enable). When BIOS_CNTL/OSWGE is set to 1, attempts to write enable the BIOS by setting BIOS_CNTL/OSWGE to 1 will immediately generate a System Management Interrupt (SMI). It is the job of this SMI to determine whether or not it is permissible to write enable to the BIOS. If it is not, it immediately sets BIOS_CNTL/OSWGE back to 0, the end result being that the BIOS is not writeable."

However, it has been shown that a race condition exists that can allow writes to the BIOS to occur between the moment that an attempt is made to set BIOS_CNTL/OSWGE to 1 and the moment that it is set back to 0 by the SMI.

Impact
A local, authenticated attacker could write malicious code to the platform firmware. Additionally, if the "UEFI Variable" region of the SPI Flash relies on BIOS_CNTL/OSWGE for write protection, as many implementations do, this vulnerability could be used to bypass UEFI Secure Boot. Lastly, the attacker could corrupt the platform firmware and cause the system to become inoperable.
Threats and Mitigation Guidelines

Secure Boot

• Disable CSM

• Set image verification defaults to secure values:
  • DENY_EXECUTE_ON_SECURITY_VIOLATION
  • QUERY_USER_ON_SECURITY_VIOLATION

• Disallow fallback to legacy boot

• Store all Secure Boot management variables as Authenticated Variables in protected flash

• Require User-Presence to disable Secure Boot

• Protect variables containing user-settings for CSM and Secure Boot Enable from unauthorized writes
Threats and Mitigation Guidelines

Key areas for concern

• Firmware Flash Regions
• UEFI Variables in Flash
• Capsule Updates
• SMM
• Secure Boot
• Option ROMs
Option ROM Exploit

- Install malware into the OS startup sequence
- Hook OS services to capture and leak sensitive data
- Hook OS services to hide from OS-level antivirus scans and other detection measures
- With legacy boot, could pollute the MBR. With UEFI boot, could replace the OS loader (Bootx64.efi, BootIA32.efi)
- Perform *persistent* destructive behavior
- Install malware into hardware devices: hard drives, USB, Thunderbolt, etc.
- Reinstall OS-level malware on reset if it was detected and removed
- Survives OS reinstall and hard drive reformat if installed in a physical OpROM (e.g., addin card)
Option ROM Exploit Limitations

• Maliciously modified ROMs should not be dispatched if Secure Boot is properly enabled
• Cannot directly infect SMM if SMRAM is already locked
• Cannot write to SPI flash if flash write protections are already enabled
• THEREFORE, should be limited to Ring 0 privileges
Additional Concerns
In addition to standard software security threats, UEFI Platform Firmware is also susceptible to additional threats, such as:

- Remote management control interfaces
- Debug hardware interfaces
- Custom security-related code implementations
- Development-oriented debugging code paths
- ASSERTs
- Password Handling
- Source code overrides
Remote Management Control

- Ensure that the most recent version of Management Engine (ME) or similar firmware is used

- Provide an easy method for end-users to update product firmware
Debug Hardware Interfaces

• Ensure that all hardware debug interfaces are disabled and locked for shipping products

• Ensure that all debug code that reports incoming/outgoing data for development is removed from shipping products

• Ensure that End of Manufacturing write-once registers and fuses are properly set/blown

https://lab.dsst.io/slides/33c3/8069.html
Custom Security-Related Code

- Always only use approved security-related algorithms and industry vetted library functions
- Never write custom security-related code even as a temporary solution, because it could end up in shipping products
Development Debugging Code

• If you’re adding debugging code that would create a vulnerability if shipped – stop and rethink! There’s most likely a better way.

• If you absolutely must add insecure code for debugging
  • Make it runtime dependent on a behavior-specific symbol so it is focused and easy to remove
  • Make definition of the runtime symbol build-dependent on the Debug-build symbol so Release builds will break
  • Clearly comment it with a specific tag and remove it as soon as possible

```c
// BUGBUG_SECURITY: define symbols to include insecure code
// for debugging purposes. Remove prior to release!
#if !defined(MDEPKG_NDEBUG)
    BOOLEAN mBypassVerification = TRUE;
    BOOLEAN mLogPassword = TRUE;
#endif

int ValidatePassword (IN CHAR16 *Password, IN UINTN Length, IN UINT8 *Hash)
{
    UINT8 PassHash [32];

    // BUGBUG_SECURITY: insecure code.
    if (mLogPassword) WritePasswordToLog (Password, Length);

    // Call a secure hashing function.
    Sha256Hash ((VOID*)Password, Length*sizeof(CHAR16), PassHash);

    // BUGBUG_SECURITY: insecure code.
    if (mBypassVerification) return 0;

    return MemCompare (Hash, PassHash, sizeof (PassHash));
}
```
**ASSERTs**

- ASSERTs are DANGEROUS, and should be avoided
- ASSERTs are compiled out of Release builds
- ASSERTs are for catching bugs that should never happen
- ASSERTs are not for catching possible errors or validating inputs
- ASSERTs used for input validation can allow for buffer overruns and other exploitable vulnerabilities

```c
EFI_STATUS TransferData(
    IN CHAR8 *InBuffer,
    IN UINT32 Length,
    IN UINT8 Id
)
{
    EFI_STATUS Status;
    UINT8 *StageBuffer;

    // Validate input parameters.
    if (InBuffer == NULL) return EFI_INVALID_PARAMETER;
    if (Length == 0) return EFI_INVALID_PARAMETER;
    if (Length > CONFIG_MAX_DATA_SIZE) return EFI_BAD_BUFFER_SIZE;

    // Create local staging buffer.
    Status = gBS->AllocatePool(EfiRuntimeServicesData, Length, &StageBuffer);
    if (EFI_ERROR (Status)) return Status;
    ASSERT (StageBuffer != NULL); // ptr should never be null if AllocatePool returns success.

    CopyMem (StageBuffer, InBuffer, Length);
    Status = TransferDataToDevice (StageBuffer, Length, Id);

    return Status;
}
```
Password Handling

- Never store passwords as raw text
- Always use an approved hashing algorithm and only store representations of passwords when needed
- Always explicitly clear buffers used to operate on passwords as soon as possible and before deallocation

```c
EFI_STATUS AuthorizeUser (VOID)
{
    EFI_STATUS Status;
    SHA256_HASH PassHash, StoredHash;
    UINT16 *Password;
    int CmpValue;

    // Get the stored representation of the password if set.
    Status = GetPassHashFromStorage (&StoredHash);
    if (EFI_ERROR (Status)) return Status; // no password set.

    Status = gBS->AllocatePool (EfiBootServicesData, MAX_PASS_SIZE, &Password);
    if (EFI_ERROR (Status)) return Status;

    // Get raw password text from user.
    Status = GetPasswordFromUser (Password, MAX_PASS_SIZE);
    if (EFI_ERROR (Status)) return Status;

    Sha256Hash ((VOID*)Password, Length*sizeof(CHAR16), &PassHash);
    ZeroMem (Password, MAX_PASS_SIZE); // explicitly clear password before deallocating buffer.
    gBS->FreePool (Password);

    CmpValue = MemCompare (&StoredHash, &PassHash, sizeof (PassHash));
    if (CmpValue != 0) return EFI_ACCESS_DENIED;
    return EFI_SUCCESS;
}
```
Overrides

Platform code often overrides portions of the core in an Override folder

- Never assume that override code contains all current security fixes to the core versions
- Always compare the override versions with the latest core versions to ensure that all security fixes are applied
- When adding custom code that could potentially add a vulnerability, always have the code security-reviewed
Validation Guidelines
Validation Guidelines

For complex systems, “Bug-Free” does not exist!

Bugs provide a means to compromise a system!
Validation Guidelines

Challenges of developing “Bug-Free” Firmware

• There are thousands and thousands of lines of code
  ➢ Manual review of all code and code paths is impractical

• There are multiple settings that must all be configured properly
  ➢ Test case matrixes for all use-cases can be overwhelming

• Even widely-accepted “safe” code can be found vulnerable
  ➢ OpenSSL 1.0.1 through 1.0.1f (Heartbleed)

• Systems rarely use the most current and secure code base
  ➢ Last minute code changes to products nearing release are risky
Validation Guidelines

Targeted Source Code Reviews

• Variable Usage and Organization
  • What would happen if a variable were deleted?
  • What benefit could an attacker gain by modifying a variable?
  • Does a variable need to be accessible at Runtime? Does it need to be modified at Runtime?
Targeted Source Code Reviews

• External Facing Code and SMI Handlers
  • Does the code properly validate externally provided input parameters? Does it use copies to prevent TOCTOU vulnerabilities?
  • Can an untrusted source provide input parameters that would cause unexpected behavior?
  • Can the code be tricked into copying data into or out of unintended address space such as SMRAM?
Validation Guidelines

Targeted Source Code Reviews

• Security Related Code
  • Are industry vetted security algorithms being used, (no custom or ad hoc implementations)?
  • Are security algorithms being used correctly?
  • Are the most recent versions of security libraries being used?
  • Are the standard core implementations being used, (no older or custom versions in an Override folder)?
  • Are there any bugs or code paths that could allow bypass of a security check?
Validation Guidelines

Validation Tools

- When a new vulnerability is discovered, always create a test for it (if possible)
- When there are any changes to code related to a security vulnerability, always re-test for the vulnerability (if possible)
- Perform fuzz and boundary testing
- Incorporate industry standard testing tools, such as CHIPSEC and automated code analysis
Next Steps
Next Steps

What Phoenix is Doing

• Performing targeted code reviews
• Developing security test tools and integrating into our QA process
• Reviewing disclosures and guidelines, and verifying our implementations
• Back porting security fixes to previous codebases
• Working with customers to educate them on important security fixes
• Monitoring the EDK2 codebase for important security fixes
• Monitoring social media for publicly disclosed findings
• Investigating emerging specifications, such as NIST SP 800-193
Next Steps

- Everyone that provides pre-OS code, and that includes firmware Option ROM code and EFI applications, needs to follow similar steps to validate their implementations
- Become a Contributor member for access to UEFI work in progress
- Select a corporate technical security representative and have them participate with the UEFI Spec Security Sub-team
- Consider participation in the Tianocore open-source development project and its security team
- Sign up to the usrt-notify email alias via admin@uefi.org to receive urgent security notifications
- Make sure you have NDAs and arrangements to receive security notifications from silicon providers, OSVs, etc.
Thanks for attending the Fall 2017 UEFI Plugfest

For more information on the Unified EFI Forum and UEFI Specifications, visit http://www.uefi.org