*presented by*

# 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.)

# Legal Stuff

# 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

# Introduction

## 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

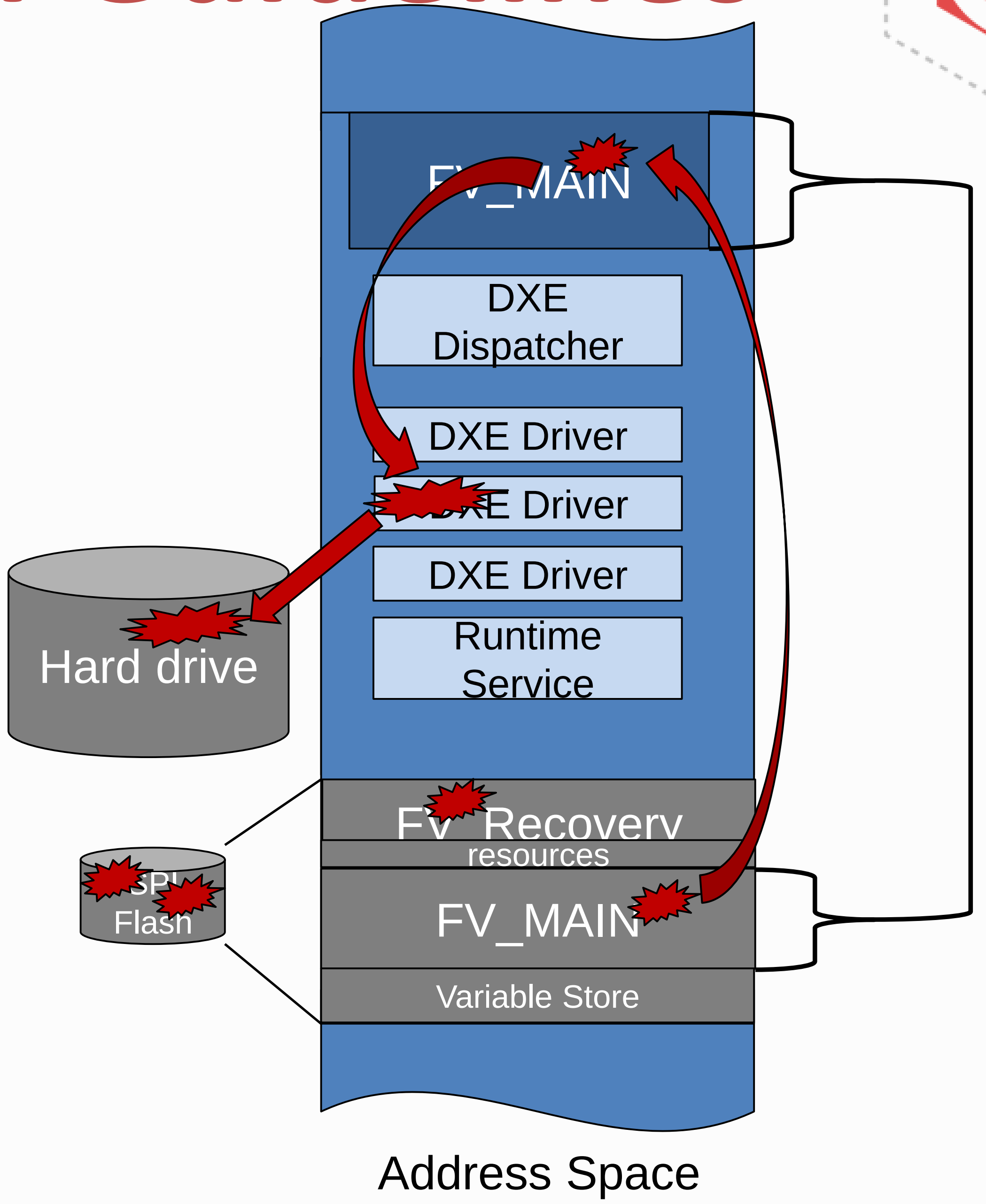
Malware injected into the **address space** is transient, and will be cleaned up on the next boot

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
- <u>Malware in flash is persistent</u>, 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

## Vulnerability Note VU#766164

Intel BIOS locking mechanism contains race condition that enables write protection bypass

Original Release date: 05 Jan 2015 | Last revised: 23 Jul 2015

Print Tweet Send Share

### Overview

A race condition exists in Intel chipsets that rely solely on the BIOS_CNTL.BIOSWE and BIOS_CNTL.BLE 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.BIOSWE and BIOS_CNTL.BLE bits as a BIOS write locking mechanism. According to Corey Kallenberg of The MITRE Corporation:

> *"When the BIOS_CNTL.BIOSWE bit is set to 1, the BIOS is made writable. Also contained with the BIOS_CNTL register is the BIOS_CNTL.s("BIOS Lock Enable"). When BIOS_CNTL.BLE is set to 1, attempts to write enable the BIOS by setting BIOS_CNTL.BIOSWE 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, and if not, immediately set BIOS_CNTL.BIOSWE back to 0; the end result being that the BIOS is not writable."*

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.BIOSWE 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.BIOSLE 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

## 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

**Vulnerability Note VU#577140**

BIOS implementations fail to properly set UEFI write protections after waking from sleep mode

Original Release date: 30 Jul 2015 | Last revised: 12 Aug 2015

Print Tweet Send Share

**Overview**

Multiple BIOS implementations fail to properly set write protections after waking from sleep, leading to the possibility of an arbitrary BIOS image reflash.

**Description**

According to Cornwell, Butterworth, Kovah, and Kallenberg, who reported the issue affecting certain Dell client systems (CVE-2015-2890):

> *There are a number of chipset mechanisms on Intel x86-based computers that provide protection of the BIOS from arbitrary reflash with attacker-controlled data. One of these is the BIOSLE and BIOSWE pair of bits found in the BIOS_CNTL register in the chipset. When the BIOSLE bit is set, the protection mechanism is enabled. The BIOS_CNTL is reset to its default value after a system reset. By default, the BIOSLE bit of the BIOS_CNTL register is cleared (disabled). The BIOS is responsible for re-enabling it after a reset. When a system goes to sleep and then wakes up, this is considered a reset from the hardware's point of view.*
>
> *Therefore, the BIOS_CNTL register must be reconfigured after waking from sleep. In a normal boot, the BIOS_CNTL is properly configured. However, in some instances BIOS makers do not properly re-set BIOS_CNTL bits upon wakeup. Therefore, an attacker is free to reflash the BIOS with an arbitrary image simply by forcing the system to go to sleep and wakes again. This bypasses the enforcement of signed updates or any other vendor mechanisms for protecting the BIOS from an arbitary reflash.*

A similar issue affecting Apple systems (CVE-2015-3692) involves the FLOCKDN bit remaining unset after waking from sleep. For more information, refer to Pedro Vilaça's blog disclosure.

**Impact**

A privileged attacker with console access can reflash the BIOS of affected systems to an arbitrary image.

# Threats and Mitigation Guidelines

## On resume from S3:

- Scripts that re-initialize the platform must be secured against malicious modifications

### Vulnerability Note VU#976132

UEFI implementations do not properly secure the EFI S3 Resume Boot Path boot script

Original Release date: 05 Jan 2015 | Last revised: 03 Aug 2015

Print Tweet Send Share

**Overview**

Some UEFI systems fail to properly restrict access to the boot script used by the EFI S3 Resume Boot Path, allowing an authenticated, local attacker to bypass various firmware write protections.

**Description**

According to Rafal Wojtczuk of Bromium and Corey Kallenberg of The MITRE Corporation:

> *"During the UEFI S3 Resume path, a boot script is interpreted to re-initialize the platform. The boot script dictates various memory and port read/write operations to facilitate this re-initialization. The boot script is interpreted early enough where important platform security mechanisms have not yet been configured. For example, BIOS_CNTL, which helps protects the platform firmware against arbitrary writes, is unlocked. TSEGMB, which protects SMRAM against DMA, is also unlocked.*
>
> *Given this, the boot script is in a security critical position and maintaining its integrity is important. However, we have discovered that on certain systems the boot script resides in unprotected memory which can be tampered with by an attacker with access to physical memory."*

**Impact**

An authenticated local attacker may be able to bypass Secure Boot and/or perform an arbitrary reflash of the platform firmware despite the presence of signed firmware update enforcement. Additionally, the attacker could arbitrarily read or write to the SMRAM region. Lastly, the attacker could corrupt the platform firmware and cause the system to become inoperable.

# 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

## Vulnerability Note VU#533140

Tianocore UEFI implementation reclaim function vulnerable to buffer overflow

Original Release date: 05 Jan 2015 | Last revised: 03 Feb 2015

Print Tweet Send Share

### Overview

The reclaim function in the Tianocore open source implementation of UEFI contains a buffer overflow vulnerability.

### Description

The open source Tianocore project provides a reference implementation of the Unified Extensible Firmware Interface (UEFI). Some commercial UEFI implementations incorporate portions of the Tianocore source code.

According to Rafal Wojtczuk of Bromium and Corey Kallenberg of The MITRE Corporation, a buffer overflow vulnerability exists in the Reclaim function. Corey Kallenberg describes the vulnerability as follows:

> *"UEFI utilizes various non-volatile variables to communicate information back and forth between the operating system and the firmware; for instance, boot order, platform language, etc. These non-volatile variables are stored in a file-system like region on the SPI flash chip. This file-system supports many operations such as deleting existing variables, creating new variables, and defragmenting the variable region in order to reclaim unused space. This latter operation is important to ensure that large variables can be created in the event the variable region is resource constrained and fragmented with many unused "free slots."*
>
> *We have discovered a buffer overflow associated with this 'reclaim' operation."*

Please note that this issue is unlikely to be directly exposed to an attacker. In order to exploit this issue, a separate vulnerability must allow prior modification of the SPI flash to enable the attacker to introduce valid variable headers after the end of the variable storage area.

### Impact

The consequences and exploitability of this bug will vary based on the particular firmware implementation. A local attacker may be able to perform an arbitrary reflash of the platform firmware and escalate privileges or perform a denial of service attack by rendering the system inoperable.

# 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

Print Tweet Send Share

### 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

# Threats and Mitigation Guidelines

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

## Vulnerability Note VU#552286

UEFI EDK2 Capsule Update vulnerabilities

Original Release date: 07 Aug 2014 | Last revised: 22 Oct 2015

Print Tweet Send Share

### Overview

The EDK2 UEFI reference implementation contains multiple vulnerabilities in the Capsule Update mechanism.

### Description

The open source EDK2 project provides a reference implementation of the Unified Extensible Firmware Interface (UEFI). Researchers at The MITRE Corporation have discovered multiple vulnerabilities in the EDK2 Capsule Update mechanism. Commercial UEFI implementations may incorporate portions of the EDK2 source code, including the vulnerable Capsule Update code.

**Buffer overflow in Capsule Processing Phase** - CVE-2014-4859
During the Drive Execution Environment (DXE) phase of the UEFI boot process, the contents of the capsule image are parsed during processing. An integer overflow vulnerability exists in the capsule processing phase that can cause the allocation of a buffer to be unexpectedly small. As a result, attacker-controlled data can be written past the bounds of the buffer.

**Write-what-where condition in Coalescing Phase** - CVE-2014-4860
During the Pre-EFI Initialization (PEI) phase of the UEFI boot process, the capsule update is coalesced into its original form. Multiple integer overflow vulnerabilities exist in the coalescing phase that can be used to trigger a write-what-where condition.

For more details, please refer to MITRE's vulnerability note.

### Impact

A local authenticated attacker may be able to execute arbitrary code with the privileges of system firmware, potentially allowing for persistent firmware level rootkits, bypassing of Secure Boot, or permanently DoS'ing the platform.

# Threats and Mitigation Guidelines

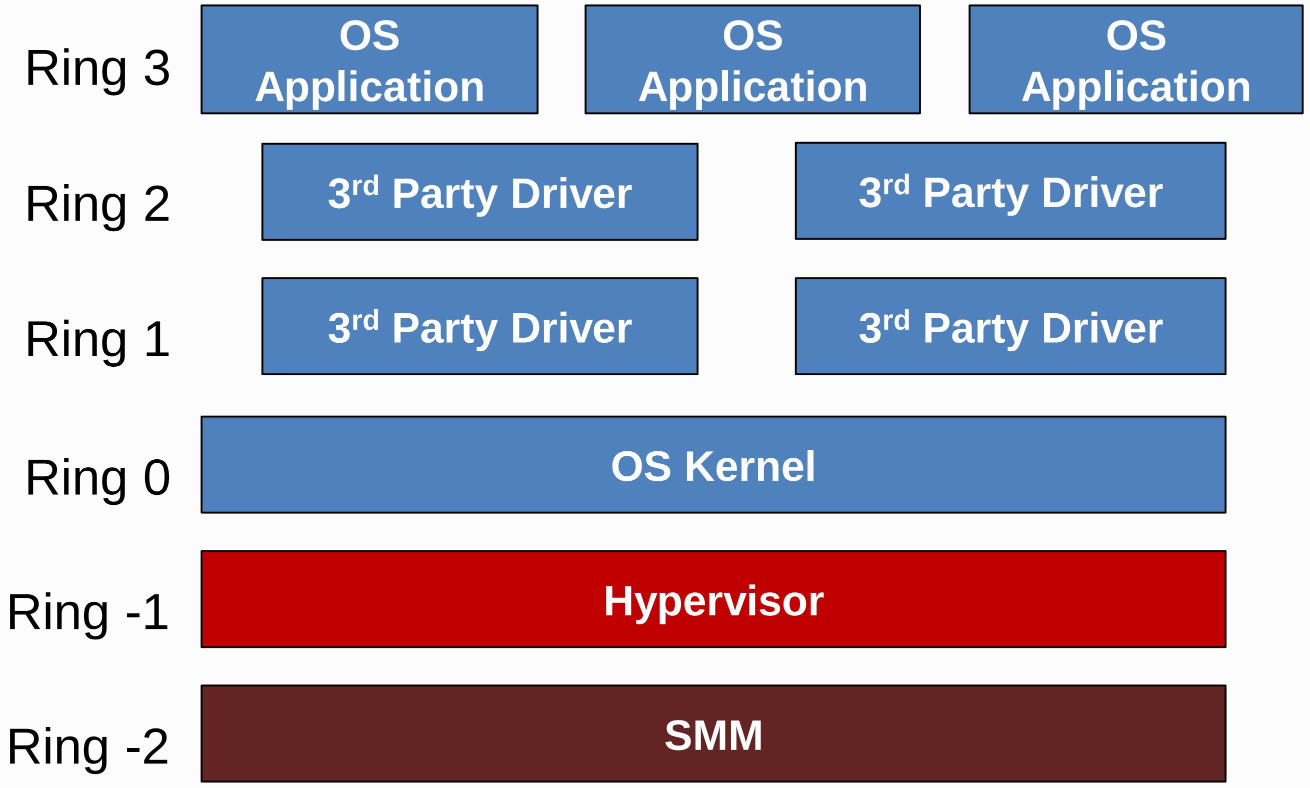
Key areas for concern

- Firmware Flash Regions
- UEFI Variables in Flash
- Capsule Updates
- SMM
- Secure Boot
- Option ROMs

# Threats and Mitigation Guidelines

## What is SMM?

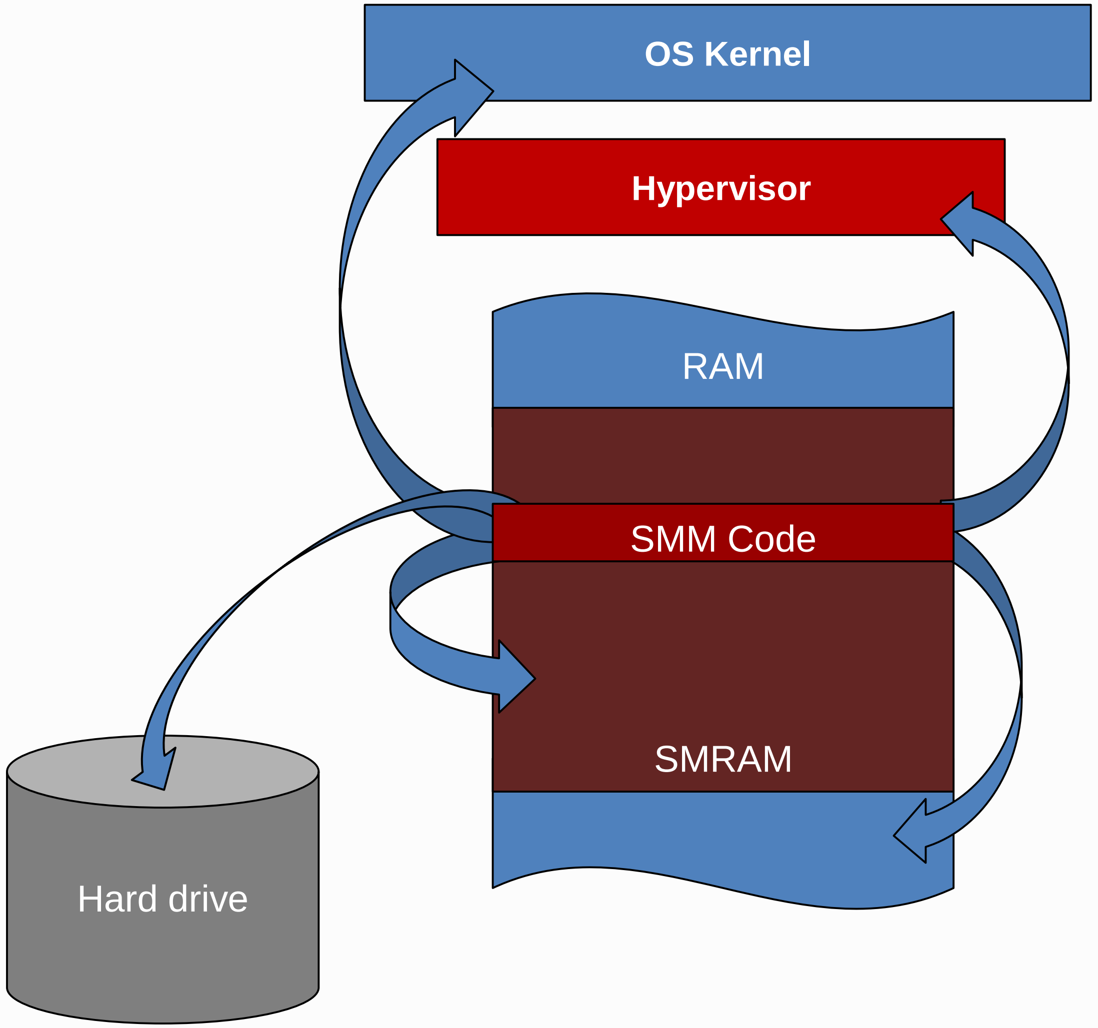
- Highly privileged processor mode
- Entered through a System Management Interrupt (SMI)
- Processor saves its context, services the SMI, then restores context and resumes
- SMM code has full visibility of all address space and devices
- Transition is transparent to the rest of the system



# Threats and Mitigation Guidelines

## SMM Mode

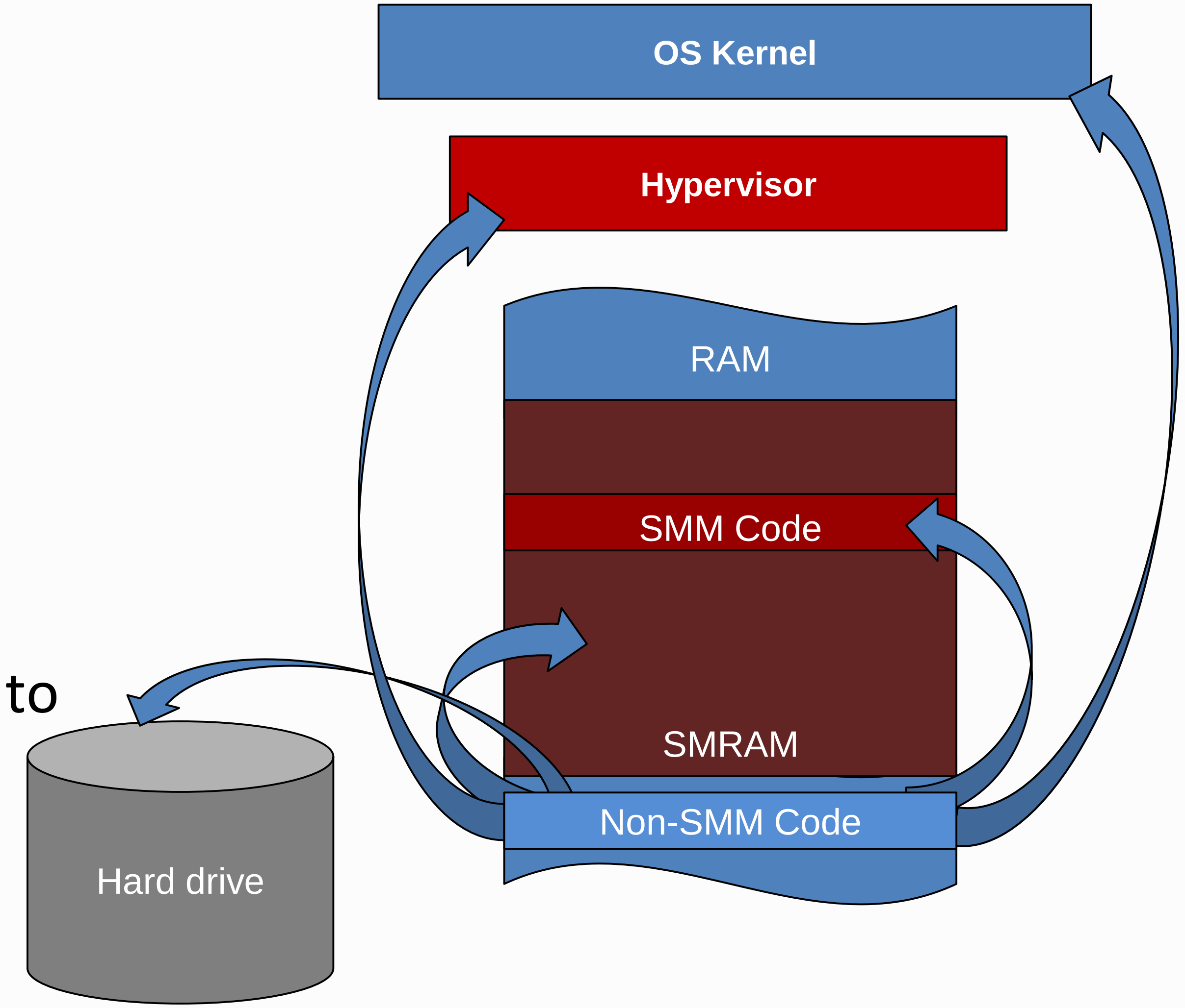
- SMM code has full access to all system memory and devices
- SMM code is not bound by OS Kernel or Hypervisor protections
- SMM code can read all of memory, modify memory contents, and even overwrite critical system files and data on storage mediums



# 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 <u>copy</u> input parameters and validate and use <u>the copy</u>, to prevent time-of-check-time-of-use (TOCTOU) vulnerabilities


**SmmRuntime Escalation of Privilege**

| | |
|---|---|
| Intel ID: | INTEL-SA-00056 |
| Product family: | Intel® Server Board S1200/1400/1600/2400/2600/4600 series |
| Impact of vulnerability: | Elevation of Privilege |
| Severity rating: | Important |
| Original release: | Aug 08, 2016 |
| Last revised: | Sep 30, 2016 |

**Summary:**

Intel is releasing mitigations for a privilege escalation issue. This issue affects the UEFI BIOS of select Intel Products. The issue identified is a method that enables malicious code to gain access to System Management Mode (SMM).

**Description:**

A malicious attacker with local administrative access can leverage the vulnerable function to gain access to System Management Mode (SMM) and take full control of the platform. Intel products that are listed below should apply the update.

Other vendors' products which use the common BIOS function SmmRuntime may be impacted. To find out whether a product you have may be vulnerable to this issue, please contact your system supplier.

# Threats and Mitigation Guidelines

## 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

### Vulnerability Note VU#631788

BIOS implementations permit unsafe SMM function calls to memory locations outside of SMRAM

Original Release date: 20 Mar 2015 | Last revised: 08 Jul 2015

Print Tweet Send Share

**Overview**

Multiple BIOS implementations permit unsafe System Management Mode (SMM) function calls to memory locations outside of SMRAM.

**Description**

Multiple BIOS implementations permit unsafe System Management Mode (SMM) function calls to memory locations outside of SMRAM. According to Corey Kallenberg of LegbaCore:

> *System Management Mode (SMM) is the most privileged execution mode on the x86 processor. Non-SMM code can neither read nor write SMRAM (SMM RAM). Hence, even a ring 0 level attacker should be unable to gain access to SMM.*
>
> *However, on modern systems, some SMM code calls or interprets function pointers located outside of SMRAM in an unsafe way. This provides opportunity for a ring 0 level attacker to break into SMM.*

In order to exploit the vulnerability, an attacker must have access to physical memory. The attacker can gain code execution in the context of SMM by first manipulating a function pointer or function called by SMM and then writing bytes to System Management Interrupt (SMI) command port `0xb2` to trigger SMM.

**Impact**

A local, authenticated attacker may be able to execute arbitrary code in the context of SMM and bypass Secure Boot. In systems that do not use protected range registers, an attacker may be able to reflash firmware.

# 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

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

Print Tweet Send Share

### 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

- SMM code must never call code outside of SMRAM as this could allow bypass of Secure Boot protections

## Vulnerability Note VU#631788

BIOS implementations permit unsafe SMM function calls to memory locations outside of SMRAM

Original Release date: 20 Mar 2015 | Last revised: 08 Jul 2015

Print Tweet Send Share

### Overview

Multiple BIOS implementations permit unsafe System Management Mode (SMM) function calls to memory locations outside of SMRAM.

### Description

Multiple BIOS implementations permit unsafe System Management Mode (SMM) function calls to memory locations outside of SMRAM. According to Corey Kallenberg of LegbaCore:

> *System Management Mode (SMM) is the most privileged execution mode on the x86 processor. Non-SMM code can neither read nor write SMRAM (SMM RAM). Hence, even a ring 0 level attacker should be unable to gain access to SMM.*
>
> *However, on modern systems, some SMM code calls or interprets function pointers located outside of SMRAM in an unsafe way. This provides opportunity for a ring 0 level attacker to break into SMM.*

In order to exploit the vulnerability, an attacker must have access to physical memory. The attacker can gain code execution in the context of SMM by first manipulating a function pointer or function called by SMM and then writing bytes to System Management Interrupt (SMI) command port `0xb2` to trigger SMM.

### Impact

A local, authenticated attacker may be able to execute arbitrary code in the context of SMM and bypass Secure Boot. In systems that do not use protected range registers, an attacker may be able to reflash firmware.

# 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

Original Release date: 05 Jan 2015 | Last revised: 23 Jul 2015

Print Tweet Send Share

### Overview

A race condition exists in Intel chipsets that rely solely on the BIOS_CNTL.BIOSWE and BIOS_CNTL.BLE 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.BIOSWE and BIOS_CNTL.BLE bits as a BIOS write locking mechanism. According to Corey Kallenberg of The MITRE Corporation:

> *"When the BIOS_CNTL.BIOSWE bit is set to 1, the BIOS is made writable. Also contained with the BIOS_CNTL register is the BIOS_CNTL.s("BIOS Lock Enable"). When BIOS_CNTL.BLE is set to 1, attempts to write enable the BIOS by setting BIOS_CNTL.BIOSWE 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, and if not, immediately set BIOS_CNTL.BIOSWE back to 0; the end result being that the BIOS is not writable."*

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.BIOSWE 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.BIOSLE 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

# Threats and Mitigation Guidelines

## 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)

# Threats and Mitigation Guidelines

## 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


39

# 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

## Vulnerability Note VU#491375

Intel Active Management Technology (AMT) does not properly enforce access control

Original Release date: 02 May 2017 | Last revised: 05 Jun 2017

Print Tweet Send Share

### Overview

Technologies based on Intel Active Management Technology may be vulnerable to remote privilege escalation, which may allow a remote, unauthenticated attacker to execute arbitrary code on the system.

### Description

CWE-284: **Improper Access Control** - CVE-2017-5689

Intel offers a number of hardware-based remote management technologies meant for maintenance of computer systems. These technologies include Intel® Active Management Technology (AMT), Intel® Small Business Technology (SBT), and Intel® Standard Manageability, and the Intel Management Engine.

These technologies listen for remote commands on several known ports. Intel's documentation provides that ports 16992 and 16993 allow web GUI interaction with AMT. Other ports that may be used by AMT include 16994 and 16995, and 623 and 664.

The Intel Management Engine that supports these technologies is vulnerable to a privilege escalation that allows an unauthenticated attacker to gain access to the remote management features provided by the Intel Management Engine. Intel has released a security advisory as well as a mitigation guide with more details.

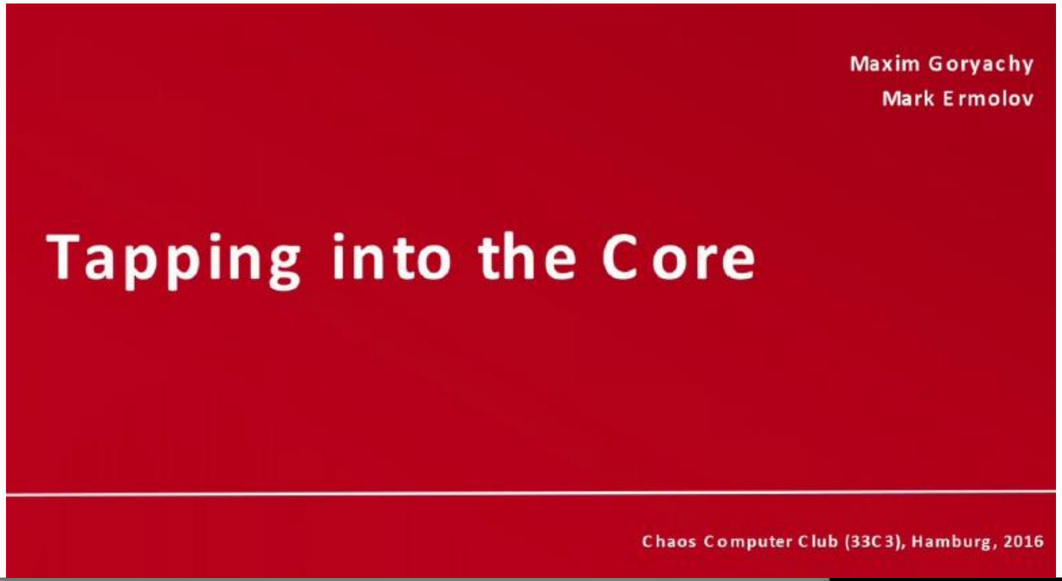
It is currently not clear how many devices or computers are shipped with Intel remote management technologies enabled by default. Original equipment manufacturers (OEMs) selling devices containing Intel products may enable remote management features by default on a model or BIOS/UEFI version basis. The CERT/CC is reaching out to OEMs to determine which if any models may be vulnerable by default. Intel's security advisory at present suggests consumer personal computers are unaffected by default. The "Vendor Information" section below contains more information.

### Impact

A remote, unauthenticated attacker may be able to gain access to the remote management features of the system. The execution occurs at a hardware system level regardless of operating system environment and configuration.

# 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

```
VOID Sha256Hash (
  IN VOID   *Password,
  IN UINTN  Length,
  OUT UINT8 *Hash
  )
{
  UINTN i;
  ZeroMem (Hash, 32);

  // Just do a simple transformation for now.
  // Replace with real code later.
  for (i=0; i<Length; i++) {
    Hash[i%32] = ((UINT8*)Password)[i] + 'W';
  }
}
```

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

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

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

# 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

```
// 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 <u>are not</u> for catching possible errors or validating inputs
- ASSERTs used for input validation can allow for buffer overruns and other exploitable vulnerabilities

```
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

```
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


48

# 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?

# Validation Guidelines

## 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


55

# 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

*presented by*


58