tianocore

EDK II Secure Code Review Guide

TABLE OF CONTENTS

EDK II Secure Code Review Guide

EDK II SECURE CODE REVIEW GUIDE

Technical Briefing

12/01/2020 06:54:44

Revision 01.0

Contributed by

Jiewen Yao, Intel Corporation

Chris Wu, Intel Corporation

Vincent J. Zimmer, Intel Corporation

Special Acknowledgements

This document checklist is collected based upon the security experience and previous security issue report. We would like to thank Sugumar Govindarajan, John Mathew, Kirk Brannock, and Karunakara Kotary of Intel Corporation who provided the thought on hardening the platform.

Acknowledgements

Revision History

Revision	Revision History	Date
01.0	Initial release.	June 2019

EDK II Secure Code Review Guide

EXECUTIVE SUMMARY

Introduction

This document describes guidelines for secure code review in EDK II firmware.

Audience

This document is intended for use by firmware developers, security reviewers, and firmware validation engineers.

GENERAL GUIDELINES FOR SECURE CODE REVIEW

Overview

Secure Code Review is a special activity compared to a normal code review. While the typical code review is focused on software quality, including usability, reusability, and maintainability, secure code reviews are focused on software security aspects, including but not limited to confidentiality, integrity, and availability (C.I.A.).

In 2006, Howard from Microsoft, published “A Process for Performing Security Code Reviews.”. It provides some general guidelines for performing a security code review. The guidelines are still valid today:

1. Make sure you know what you are doing
2. Prioritize
3. Review the code.

Make sure you know what you are doing

Before you review code, please make sure you understand the following:

1. Threat Model and Security Architecture of the feature, including assets, security objectives, adversaries, and the mitigations.
2. The general secure design and coding principles for EDK II.

Prioritize

According to “A Process for Performing Security Code Reviews.”, the priority of common software code is below:

1. Old code
2. Code that runs by default
3. Code that runs in an elevated context
4. Anonymously accessible code
5. Code listening on a globally accessible network interface
6. Code is written in C/C++/assembly language
7. Code with a history of vulnerabilities
8. Code that handles sensitive data
9. Complex code
10. Code that changes frequently

Except for items #4 and #5, all other rules apply to EDK II firmware.

Review the code

Reviewing the code involves three steps:

1. Rerun all available code-analysis tools.
2. Look for common vulnerability patterns, such as
    i. Integer arithmetic vulnerabilities
    ii. Buffer-overrun vulnerabilities
    iii. Cryptographic vulnerabilities
    iv. Structured Query Language (SQL) Injection vulnerabilities

   v. Cross-site scripting vulnerabilities
3. Dig deep into risky code, such as
   i. Are there logic or off-by-one errors (for example, '>' vs. '>=' or '||' vs. '&&')?
   ii. Is the data correctly validated?
   iii. Are buffer lengths constrained correctly?
   iv. Are integer values range-checked correctly?
   v. Are pointers validated?
   vi. Can the code become inefficient (for example, $O(N^2)$) due to some malformed data (for example, a hash table look-up becomes a list look-up)?
   vii. Are errors handled correctly?

Other than #2.iv and #2.v above, all other rules apply to EDK II firmware.

Besides "A Process for Performing Security Code Reviews.", Ransome provided some good suggestions in the book "Core Software Security: Security at the Source" on how to perform the SDL activity including security code review.

EDK II Secure Coding Guidelines

We also provided the guideline for EDK II Secure Coding. People need to fully understand the EDK II secure coding best practices before doing the security code review.

CODE REVIEW GUIDELINES FOR BOOT FIRMWARE

Based on previous analysis of firmware issues, vulnerabilities fall into 8 general categories that should be the focus of secure code reviews:

1. External Input
2. Race Conditions
3. Hardware Input
4. Secret Handling
5. Register Lock
6. Secure Configuration
7. Replay/Rollback
8. Cryptography

This section discusses each class of vulnerability and summarizes approaches for review.

External Input

External input describes data that can be controlled by an attacker. Examples include:

- UEFI capsule image
- Boot logo in Bitmap (BMP) or Joint Photographic Experts Group (JPEG) format
- Contents of file system partitions
- Read/write variables
- System Management Mode (SMM) communication buffer
- Network packets

Previous Vulnerabilities:

Boot Logo Image

At BlackHat 2009, Invisible Things Lab demonstrated how to use a buffer overflow in BMP file processing to construct an attack and flash a new firmware. The BMP file is an external input where an attacker may input a large value for `PixelWidth` and `PixelHeight`. This causes `BltBufferSize` to overflow and results in a very small number. This is a typical integer overflow caused by multiplication.

```
EFI_STATUS ConvertBmpToGopBlt ()
{
 /// ...
  if (BmpHeader->CharB != 'B' || BmpHeader->CharM != 'M') {
    return EFI_UNSUPPORTED;
  }
  BltBufferSize = BmpHeader->PixelWidth * BmpHeader->PixelHeight
                  * sizeof (EFI_GRAPHICS_OUTPUT_BLT_PIXEL);
  IsAllocated = FALSE;
  if (*GopBlt == NULL) {
    *GopBltSize = BltBufferSize;
    *GopBlt = EfiLibAllocatePool (*GopBltSize);
```

To handle these cases, code should check for integer overflow using division, as shown below:

```
if (BmpHeader-&gt;PixelWidth &gt; MAX_UINT / sizeof
(EFI_GRAPHICS_OUTPUT_BLT_PIXEL) / BmpHeader-&gt;PixelHeight) {
    return EFI_INVALID_PARAMETER;

}
```

SMM Callout

At Black Hat DC 2009, Invisible Things Lab demonstrated a way to inject code into SMM. The SMM code referenced (`ACPINV` below) a function pointer in Advanced Configuration and Power Interface (ACPI) Non-Volatile Storage (NVS) memory and invoked this function address. An attacker may modify the function pointer address in ACPI NVS so it points to a malicious function.

```
mov [ACPINV+x], %rax
call *0x18(%rax)
```

A similar issue is also found in ThinkPad 2016. The `SmmRuntimeCallHandle` is the pointer in ACPI Reserved memory. As such, the attacker may replace this function pointer with any address.

This is shown in line with the statement: `RtServices = (EFI_SMM_RT_CALLBACK_SERVICES *) SmmRtStruct->PrivateData.SmmRuntimeCallHandle;` below.

```
EFI_STATUS
EFIAPI
SmmRuntimeManagementCallback (
  IN EFI_HANDLE             SmmImageHandle,
  IN OUT VOID               *CommunicationBuffer,
  IN OUT UINTN              *SourceSize
  )
{
  SMM_RUNTIME_COMMUNICATION_STRUCTURE *SmmRtStruct;
  EFI_SMM_RT_CALLBACK_SERVICES        *RtServices;

  RtServices  = NULL;

  SmmRtStruct = (SMM_RUNTIME_COMMUNICATION_STRUCTURE *) CommunicationBuffer;
  RtServices  = (EFI_SMM_RT_CALLBACK_SERVICES *) SmmRtStruct->PrivateData.SmmRuntimeCallHandle;

  if (RtServices != NULL) {
    RtServices->CallbackFunction (RtServices->Context, mSmst, (VOID *) &SmmRtStruct->PrivateData);
    SmmRtStruct->PrivateData.SmmRuntimeCallHandle = NULL;
  }

  return EFI_SUCCESS;
}
```

It is critical that SMM never reference memory outside System Management RAM (SMRAM) for function pointers.

In the latest Intel processors, the SMM_Code_Access_Chk feature can be used to block code execution outside of the value set by the SMRAM Range Register (SMRR). This feature MUST be enabled if it is supported.

The latest versions of EDK II also enable Executable Disable (XD) for memory addresses outside of SMRAM.

SMM Communication

In CanSecWest 2015, a new class of SMM attack was disclosed. The attacker may construct a SMM communication buffer that points to memory owned by System Management RAM (SMRAM) or Virtual Machine Monitor (VMM), then pass this address into a System Management Interrupt (SMI) handler. This causes the SMI handler to perform the write for the attacker. This typically classified as a “confused deputy” attack. See the lines with `CommBuffer` and with the `CopyMem` statement below.

```
SmmVariableHandler ()
//  ...
  SmmVariableFunctionHeader = (SMM_VARIABLE_COMMUNICATE_HEADER *)CommBuffer;
  switch (SmmVariableFunctionHeader->Function) {
  case SMM_VARIABLE_FUNCTION_GET_VARIABLE:
    SmmVariableHeader = (SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE *)
    SmmVariableFunctionHeader->Data;
    Status = VariableServiceGetVariable (
               ...
               (UINT8 *)SmmVariableHeader->Name + SmmVariableHeader->NameSize
               );
}

VariableServiceGetVariable (
  // ...
  OUT VOID *Data
  )
{
 // ...
  CopyMem (Data, GetVariableDataPtr (Variable.CurrPtr), VarDataSize);
}
```

To mitigate this attack, the SMI handler is required to use the library service `SmmIsBufferOutsideSmmValid()` to check the communication buffer before accessing it.

ACPI table for Authenticated Code Module (ACM) is a signed binary module delivered by Intel. It is used to construct a dynamic root of trust for measurement (DRTM) environment. In 2011, Invisible Things Lab disclosed a way to hijack the SINIT ACM. The issue happens when the ACM code parses the untrusted ACPI DMA Remapping (DMAR) table. The DMAR table is used before validation of the address. As such the attacker may control the copied memory length and override the Intel Trusted Executable Technology (TXT) heap and SINIT ACM itself. See line `6741` below.

```
6675: mov  (%edi),%esi
6677: cmpl $0x52414d44,(%esi)
; (DWORD*)esi == 'DMAR'?

667d: je 0x6697
...
6697: mov  (%edi),%edi
6699: mov  %edi,%es:0xa57
; var_a57 = &dmar

66a0: mov  0x4(%edi),%ecx
; ecx = dmar.len

66a3: push %ecx
66a4: add  %edi,%ecx
66a6: mov  %ecx,%es:0xa5b
; var_a5b = &dmar + dmar.len

...
6701: mov  %es:0xa47,%edi
; edi = var_a47 (memory on the TXT heap)

6708: mov  (%edi),%eax
670a: mov  %es:0xa5b,%ebx
; ebx = &dmar + dmar.len

6711: sub %es:0xa57,%ebx
; ebx = dmar.len

...
6738: mov %es:0xa57,%esi
; var_a57 = &dmar

673f: mov %ebx, %ecx
6741: rep movsb %ds:(%esi),%es:(%edi)
; memcpy (var_a47, dmar, dmar.len)
```

Adding a check for the length field of untrusted data source is mandatory.

Capsule Image

Most UEFI firmware supports capsule based firmware update. In 2014, MITRE demonstrated how to use a vulnerability in the capsule coalesce process to attack the firmware update process.

This is another example of an integer overflow. **NOTE**: `MemorySize if` statement and `Size +=` below.

```
EFI_STATUS
EFIAPI
CapsuleDataCoalesce (
  IN EFI_PEI_SERVICES               **PeiServices,
  IN EFI_PHYSICAL_ADDRESS           *BlockListBuffer,
  IN MEMORY_RESOURCE_DESCRIPTOR     *MemoryResource,
  IN OUT VOID                       **MemoryBase,
  IN OUT UINTN                      *MemorySize
  )
{
```

```
  //...
    if (*MemorySize <= (CapsuleSize + DescriptorsSize)) {
      return EFI_BUFFER_TOO_SMALL;
  }
  //...
}

EFI_STATUS
GetCapsuleInfo (
  IN EFI_CAPSULE_BLOCK_DESCRIPTOR   *Desc,
  IN OUT UINTN                      *NumDescriptors OPTIONAL,
  IN OUT UINTN                      *CapsuleSize OPTIONAL,
  IN OUT UINTN                      *CapsuleNumber OPTIONAL
  )
{
//  ...
    } else {
      Size += (UINTN) Desc->Length;
      Count++;
  ...
}
```

Before the code performs the addition, the code must use subtraction to check if the addition will cause an integer overflow.

Read/Write Variable

A read/write variable is another potential attack surface because it is easily controlled by an attacker. In CanSecWest 2014, MITRE demonstrated how to modify the “Setup” variable to bypass UEFI secure boot ImageVerificationPolicy.

The attack taught us that it is a bad idea to embed security policy in a read/write “Setup” variable.

S3 Boot Script

The S3 Boot Script is used to restore the register settings during the ACPI S3 resume process. In CanSecWest 2015, Invisible Things Lab found some firmware implementations did not protect the S3 script or the dispatch function code, so it remained in an OS-accessible ACPI memory region. This allowed an attacker to inject malicious boot script content to bypass the silicon lock register setting in the S3 Boot Script.
See the use of `EntryFunc` and `EntryPoint` below.

```
BootScriptExecuteDispatch (IN UINT8 *Script)
{
   ...
   EntryFunc = (DISPATCH_ENTRYPOINT_FUNC) (UINTN) (ScriptDispatch.EntryPoint);
   Status = EntryFunc (NULL, NULL);
}
```

As a mitigation, the lockbox should be used to protect data used in the S3 resume phase.

Network for AMT

Intel® Active Management Technology (Intel® AMT) is a remote management feature in the Intel vPRO platform. In 2017, Embed disclosed an issue with Intel AMT where providing an empty response will cause password verification to succeed as if the attacker provided the admin password. See the use of `strncmp` and `response.length` below.

```
/* NETSTACK_CODE:20431FC8 */

if(strncmp(computed_response, response.value, response.length))
```

```
{
   goto error;
}
return 0;
```

To avoid similar issues, network packet processing code should always be carefully reviewed.

Race Condition

There are two typical race conditions found in firmware:

1. Race condition in a data buffer
2. Race condition in a register unlocking mechanism.

Previous Vulnerabilities:

Race condition for data buffer

The typical example is the SMM communication buffer. If the check function verified the non-SMRAM copy of communication buffer and then uses it, the attacker may use another CPU thread to perform Time-of-Check/Time-of-Use (TOC/TOU) attack to modify the buffer content after it is checked.

To mitigate this, the communication buffer must be copied into SMRAM before it is checked.

Another example is the motherboard flash content. When Intel Boot Guard is enabled, the Authenticated Code Module (ACM) loads Initial Boot Block (IBB) flash into cache and validates the cached copy. An attacker may use the flash programmer to update the IBB flash copy after it is loaded by ACM. This is a variation of a Time-of-Check/Time-of-Use attack.

The IBB cache copy mechanism needs to ensure that no code or data in the IBB flash can be referenced.

Race condition for register unlock

In 2014, MITRE found a race condition, named Speed Racer, which allows an attacker to subvert a component of the firmware flash protection mechanisms.

Secure code review must verify that SMM code does not leave threads outside of SMRAM when there is flash protection is in an unlocked state.

Hardware Input

Hardware input is a special class of external input. If an attacker controls hardware, the input from hardware is considered to be untrusted. This includes, but is not limited to, Memory Mapped Input/Output (MMIO), cache, Direct Memory Access (DMA), Universal Serial Bus (USB) descriptors, and Bluetooth Low Energy (BLE) advertisement data.

Previous Vulnerabilities:

MMIO BAR Overlap

In BlackHat 2008, Invisible Things Lab demonstrated how to program the remap Base Address Register (BAR) to make the remap memory overlap with VMM or SMRAM, thus allowing for subsequent modification of the VMM or SMRAM contents.

```
pci_write_word (dev, TOUUD_OFFSET, (new_remap_limit+1)&amp;lt;&amp;lt;6);
pci_write_word (dev, REMAP_BASE_OFFSET, new_remap_base);
pci_write_word (dev, REMAP_LIMIT_OFFSET, new_remap_limit);
```

In BlackHat 2009, Invisible Tings Lab also found the remap register bar can make the remap memory overlap with Management Engine (ME) RAM, thus allowing for a modification of the contents in ME firmware.

To mitigate this class of attack, verify register bars are properly locked

MMIO BAR Access

In RECon 2017, Intel disclosed the MMIO BAR access issue in SMM. The attacker may configure the MMIO BAR to make it overlap with SMRAM. After this, subsequent access to MMIO in SMM becomes accesses to SMRAM. See statements with `bar` assignment within `if` statement below.

```
static void mainboard_smi_brightness_down (void)
{
  u8 *bar;
  if ((bar = (u8 *)pci_read_config32(PCI_DEV(1, 0, 0), 0x18))) {
    printk(BIOS_DEBUG, “bar: %08X, level %02X\n”, (unsigned int)bar,
    *(bar+LVTMA_BL_MOD_LEVEL) &amp;amp;= 0xf0;
    if (*(bar+LVTMA_BL_MOD_LEVEL) &amp;gt; 0x10)
      *(bar+LVTMA_BL_MOD_LEVEL) -= 0x10;
  }
}
```

There are several ways for firmware to mitigate this class of attack. For example, SMM can verify the MMIO bar does not overlap with SMRAM or is not in DRAM before access. SMM can revert the MMIO bar value to the default setting, perform an operation, then restore it to the original value.

Care must be taken when code checks the MMIO. In 2009, Invisible Things Lab showed an incorrect check for MMIO BAR. This code checks the Memory Controller Hub (MCH) BAR value, but only for the lower 32 bits. Since the MCH BAR is 36 bits, the attacker may configure the MCH BAR value above 4G and exploit ACM due to the error in validation. This can results in an improper setup for the Intel® Virtualization Technology for Direct I/O (Intel® VT-d) engine. See the usage of `MCHBAR address` below

```
pusha
mov eax, 0x48 ; MCHBAR address
call pci_get_long
and ebx, 0xfffffffe
```

```
mov DWORD PTR es:MCHBAR, ebx
cmp ebx, 0xfec04000
ja continue
mov al, 0x4
mov ah, 0xc
call sinit_error
continue:
or ebx, 0x1
call pci_write_long
popa
ret
```

Cache

In CanSecWest 2009, Cache poisoning was used to attack SMRAM in 2009. The attacker modifies the Memory Type Range Register (MTRR) to make it overlap with SMRAM, then updates the SMRAM cache and triggers an SMI.

Recent Intel processors have introduced the SMRAM Range Register (SMRR) to resist cache poison attack. SMRR must be setup for all logical processors. This prevents the MTRR overlap with SMRAM from taking effect.

DMA

In BlackHat 2013, the NCC group demonstrated a DMA attack using Thunderbolt. In 2017, OS password theft was demonstrated using PCIleech hardware.

DMA attacks can be mitigated by setting up the Input/Output Management Unit (IOMMU) to block DMA access to full system memory. In firmware, this can be achieved using the IOMMU or disabling the Peripheral Component Interconnection (PCI) Bus Master Enable (BME) bit. However, if an untrusted device driver requires PCI BME access, the IOMMU must be setup to accommodate the untrusted device.

USB

Because attackers can create devices with bad USB descriptors, USB data is considered untrusted. Projects like Facedancer are good examples of USB fuzzing tools. In BlackHat 2014, a demo shows how to do fuzz for the USB device driver.

USB firmware drivers must assume USB descriptors are untrustworthy and always verify before consumption. This policy should also be applied to other drivers that consume potentially untrustworthy data, such as Bluetooth device advertisement messages.

TPM Genie

In 2018, the NCC group demonstrated that a Trusted Platform Module (TPM) Genie may cause memory corruption in different TPM stacks, including Linux, tboot, and UEFI. This is possible when data returned by the TPM is not validated by the TPM stack. See the usage of `recd` in the statements below.

```
int tpm_get_random(u32 chip_num, u8 *out, size_t max) {
  struct tpm_chip *chip;
  struct tpm_cmd_t tpm_cmd;
  u32 recd, num_bytes = min_t(u32, max, TPM_MAX_RNG_DATA);
  ...
  tpm_cmd.header.in = tpm_getrandom_header;
  tpm_cmd.params.getrandom_in.num_bytes = cpu_to_be32(num_bytes);
  err = tpm_transmit_cmd( chip, &amp;tpm_cmd,
  TPM_GETRANDOM_RESULT_SIZE + num_bytes );
  ...
  recd = be32_to_cpu(tpm_cmd.params.getrandom_out.rng_data_len);
  memcpy(out, tpm_cmd.params.getrandom_out.rng_data, recd);
  ...
```

```
}
```

As mitigation, the TPM driver must perform robust checks of the response buffer size.

Secret Handling

In some cases, the users are required to input passwords in the firmware, such as setup administrator password, hard drive password, and Trusted Computing Group (TCG) OPAL password. Sometimes the firmware also includes some password or access key. We need a good way to handle these secrets.

Previous Vulnerabilities:

Password not cleared in memory

In DefCon 2008, iViZ disclosed a way to get the password from the BIOS Data Area (BDA) because the BIOS does not clear the keyboard buffer which contains the password information.

After the password is used, the code should always clear it in its various locations: input key buffer, stack, heap, global variable, etc.

Key based protection

In BlackHat 2019, Mastrov disclosed how to brute force search Computrace disable key in SMRAM. The key comparison algorithm does not have a constant time. Also, the final key is only 1 byte. See the statement using `key_match` below.

```
key_byte = cpu_regs->EBX;
ComputraceState.Active = TRUE;
ComputraceState.DisableSecreteKey[0] = key_byte & 0xff;
ComputraceState.DisableSecreteKey[1] = (key_byte & 0xff00) >> 8;
ComputraceState.DisableSecreteKey[2] = (key_byte & 0xff0000) >> 16;
ComputraceState.DisableSecreteKey[3] = (key_byte & 0xff000000) >> 24;

key_match = TRUE;
for (i = 0; i < 4; i) {
  if (key[i] != ComputraceState.DisableKey[i]) {
    key_match = FALSE;
    break;
  }
}
```

This is a vulnerable inside channel attack. The duration of the verification then reveals the index of the character. The code should always use a mechanism that compares the entire data before completion. Note a single-byte key is vulnerable to brute force attack.

Default key

In BlackHat 2011, Accuvant Lab disclosed a way to access battery firmware because the access key is unchanged. The below disassembly code shows the 0x36720414 is hardcoded. It is also the default unseal key in a public document. See statements below moving constants into `edx`

```
UnSeal_LSW:
xor eax, eax
mov edx, 0414h
call writeSBWord
test eax, eax
jz short UnSeal_MSW
...
UnSeal_MSW:
xor eax, eax
mov edx, 3672h
call writeSBWord
test eax, eax
```

```
jz short loc_26FD
```

The vendor should always change the default password or key for a device to prevent illegal access. Also, it is not a good idea to hardcode the key in the source code.

Another example in TPM2, during boot, the platform should always send Tpm2HierarchyChangeAuth(TPM_RH_PLATFORM) command to a TPM2 device to prevent other code accessing the TPM2 platform hierarchy. The same action must be done in S3 resume too.

Register Lock

When the system powers on, most of the silicon registers are unlocked. The firmware code needs to configure the system and lock the critical resources by setting the lock bit in a silicon register. Examples include but are not limited to flash chip lock, SMM lock, SMI lock, MMIO BAR configuration lock, Model Specific Register (MSR) configuration lock, etc.

Previous Vulnerabilities:

Flash

In 1998, older platforms did not properly lock access to the flash parts, allowing anyone to overwrite BIOS code. Sixty million computers were believed to be infected by the CIH) virus.

In Power Of Community 2007, a new attack appeared which took advantage of the Intel top swap feature, if the latter capability was unlocked.

Today, there are several ways to lock the flash part, and the firmware should lock all the possible ways, in proper time, and in all boot paths. These paths include a normal boot, S3, S4, capsule update, recovery, etc.

SMRAM

It is likely the first documented SMM attack, which occurred because the SMM memory range was not locked.

Platforms must lock SMRAM in silicon and setup SMRR for all processors to protect SMRAM. This lock must happen in all boot paths (normal boot, S3, S4, capsule update, recovery, etc.).

MMIO BAR

In BlackHat 2008, Invisible lab demonstrated how to use unlocked remap registers for SMM or Management Engine (ME) firmware to inject code.

Today, all critical MMIO bars are required to be locked without overlap. The configuration is checked by the ACM during a TXT DRTM launch.

Secure Configuration

For security features, it is not a good idea to use variables to control the behavior because they can be altered by an attacker to bypass protection. The general configuration also includes the system state, memory configuration, different boot mode, etc.

Previous Vulnerabilities:

UEFI Secure Boot

In CanSecWest 2014, MITRE disclosed the vulnerability that the OEM used setup a variable to control the image verification policy. That meant the UEFI secure boot could be easily bypassed. See lines below assigning values to `policy` within each `case` statement.

```
DxeImageVerificationHandler(EFI_EXECUTABLE Image) {
  switch (getImageOrigin(image)) {
  case IMAGE_FROM_OPTION_ROM:
    policy = Setup.LOAD_FROM_OROM;
  case IMAGE_FROM_FIXED_DRIVE:
    policy = Setup.LOAD_FROM_FIXED;
  case IMAGE_FROM_REMOVABLE:
    policy = Setup.LOAD_FROM_REMOVABLE;
  ...
  if (policy == ALWAYS_EXECUTE)
    return EFI_SUCCESS;
  else
    return IsImageAllowed(image);
}
```

For any security feature, there should be no way to bypass it in the production. No variable should be used to control it. If a Platform Configuration Database (PCD) is used, the PCD must be statically configured.

Intel® Boot Guard

In 2016 and DefCon 2017, Ermolov disclosed how to bypass Intel® Boot Guard.

In BlackHat 2017 and BlackHat 2019, Mastrov disclosed how to bypass Intel® Boot Guard. See lines below assigning values to `BootGuardVerifyTransitionPEItoDXEFlag` followed by a check.

```
EFI_STATUS BootGuardPei (EFI_PEI_SERVICES **PeiServices, VOID *Ppt)
{
  ...
  if (!((BootGuardHashKeySegment1 == 0) {
    CalculateSha256 (BootGuardHashKeySegment1);
    CalculateSha256 (CurrentBootGuardHashKey1);
    if (!MemCmp (BootGuardHashKeySegment1, CurrentBootGuardHashKey1, 32)) {
      BootGuardVerifyTransitionPEItoDXEFlag = 1;
    } else {
      BootGuardVerifyTransitionPEItoDXEFlag = 0;
      return EFI_SUCCESS;
    }
  }
  return Status;
}

EFI_STATUS BootGuardDxe (EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
  ...
  if (BootGuardVerifyTransitionPEItoDXEFlag == 0) {
    BootGuardRegisterCallback();
  }
```

```
  return EFI_SUCCESS;
}
```

The summary of the issue is below:

1. The Intel® Boot Guard configuration is not set properly.
2. The verification does not always happen in all boot modes. For example, the verification is done only once every 12 times a device is powered up.
3. The software logic issue in Intel Boot Guard PEI or DXE that the verification may be bypassed in some cases.

The mitigation is:

1. Fuse configuration – always verify the fuses are configured for security.
2. Verification – ensure that verification occurs in all boot modes and boot paths.

TCG Trusted Boot

In BlackHat 2018, Han disclosed an issue about TPM measurements in a DRTM environment. This issue was related to the S3 resume path, where TBOOT only measured code and read-only data for the Measured Launch Environment (MLE). However, TBOOT did not measure the required initialized data. This created a condition where an attacker could hijack the control flow and exploit TBOOT. See lines below with statements `_mle_end` and `.data`.

```
_mle_start = .;              /* beginning of MLE pages */
  *(.text)
  *(.fixup)
  *(.gnu.warning)
  } :text = 0x9090

.rodata : { *(.rodata) *(.rodata.*) }
. = ALIGN(4096);

_mle_end = .;                /* end of MLE pages */

.data : {           /* Data */
  *(.data)
  *(.tboot_shared)
  CONSTRUCTORS
  }
```

Mitigation occurs when MLE sets up the environment, ensuring that all critical data (code, read-only data, and initialized data) is measured, including the function pointers. This demonstrates the importance of a complete measurement.

In BlackHat 2019, Han disclosed an issue using TPM in a static root-of-trust for measurement (SRTM) environment. During the S3 resume path, if the OS does not send Shutdown(STATE) the firmware Startup(STATE) will fail. Some platform firmware only sent Startup(CLEAR) which left all Platform Configuration Registers (PCR) open. See lines below with if statement `BootMode == BOOT_ON_S3_RESUME` and then `Status = Tpm2Startup (TPM_SU_CLEAR);` statement.

```
PeimEntryMA ()
{
  if (BootMode == BOOT_ON_S3_RESUME) {
    Status = Tpm2Startup (TPM_SU_STATE);
    if (EFI_ERROR (Status) ) {
      Status = Tpm2Startup (TPM_SU_CLEAR);
```

The mitigation extends the PCR with an EV_SEPARATOR error, which takes advantage of proper error handling.

Replay/Rollback

Replay is the ability to use a previously used credential that was designed for one-time approval to access protected content beyond the first instance. Typically, a timestamp, nonce value, or monotonic counter can be used to detect replay.

Rollback is the ability to start at a newer level of a release and go back to a forbidden earlier level of a release. Typically, the firmware needs to use a lowest support version (LSV) or secure version number (SVN) to control the update.

Cryptography

Cryptography is also an indicator we need to consider when we design a proper solution. Choosing the right cryptographic algorithm is important. A checksum or CRC value is no longer considered to be strong protection. Cryptographic key management must be considered as part of a complete security solution.

Previous Vulnerabilities:

In BlackHat 2009, Chen demonstrated how to add a rootkit to Apple Keyboard firmware via a firmware update.

In 2010, Weinmann demonstrated how to add a rootkit to ThinkPad embedded controller (EC) firmware via update.

In 2011, Cui demonstrated how to add a rootkit to HP printer firmware via update.

All of the cases above demonstrate the need for firmware locking and authenticated updates.

Other

As a final note, firmware must not contain any “back door” access mechanisms. Attackers have experience in reverse engineering, making back doors easy to detect. This is especially important for code related to SMI handlers, UEFI variables, or key management.

In BlackHat 2018, Domas demonstrated how to find a hidden instruction to gain supervisor privileges in user mode. He used fuzzing to scan the system and found a special “God Mode Bit” (MSR 1107, BIT 0). Toggling this bit activated a launch instruction (0F03). By using a co-located core with unrestricted access to the core register file, software can send content via Ring3 to modify a Ring0 register and obtain hardware privilege escalation.

Summary

Category	Review Detail
External Input	What is the external input? How is the external input checked? Does the check happen in all possible paths? What is the action if the check failed? If SMM is involved, how does SMI handler do the check for the communication buffer? If a Variable is involved, how is it consumed? Is ASSERT used?
Race Condition	What is the critical resource? If SMM is involved, can the BSP and AP access the same resource? Does the trusted region code access resources in the untrusted region?
Hardware Input	What is the hardware input? How is the hardware input checked? Does the check happen in all possible paths? If MMIO is involved, how is the MMIO bar checked?
Secret Handling	Where is the secret? How is the secret cleared after use? Does the cleanup function clear all secrets in all places, such as stack, heap, global data, communication buffer, ASCII < = > Unicode, Setup Browser, Key buffer? Is the secret saved into a variable? Does the password follow the general rules, such as strong password requirement, retry time, history, etc? What if the user forgets the password? Is the default password/key used? Is the password/key hardcoded? Does the key comparison algorithm compare entire data? Is side channel guidelines followed?
Register Lock	What registers need to be locked? When is the register locked? Is the register lock controlled by some policy? Is the register lock controlled by a variable? Is there any way to bypass the lock? Is the register locked in normal boot, S3, S4? Is the register locked in capsule, recovery? Is the register locked in manufacture mode?
Secure Configuration	Is a variable used to control the policy? Is a PCD used to control the policy? If so, what is the PCD type? What is the default configuration? What is the behavior in S3, S4, capsule, recovery, manufacture mode or debug mode?
Replay/Rollback	Is LSV or SVN used? Where is the LSV or SVN stored? How are timestamps, nonce, or monotonic counters used?
Cryptograph	Is a signing verification algorithm used? Is a deprecated algorithm used? Is Cyclic Redundancy Check (CRC) or checksum used? Should the solution use hash or Hashed Message Authentication Code (HMAC)? Should the solution use symmetric encryption or asymmetric encryption? When is the key deployed and destroyed? Where is the key located? How is the key protected? Is the key root key or session key used to encrypt the data?

REFERENCES

Books and Papers

[Cohen] Best Kept Secrets of Peer Code Review, Jason Cohen, Smart Bear Inc., 2006, ISBN: 978-1599160672

[Freedman] Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products, Daniel P. Freedman and Gerald M. Weinberg, Dorset House, 1990, ISBN: 978-0932633194

[Gilb] Software Inspection, Tom Gilb and Dorothy Graham, Addison-Wesley Professional, 1994, ISBN: 978-0201631814

[Howard] Howard, M. (2006, July–August). “A Process for Performing Security Code Reviews.” IEEE Security & Privacy, pp. 74–79, https://www.researchgate.net/publication/3437819_A_process_for_performing_security_code_reviews?ev=auth_pub

[Ransome] Core Software Security: Security at the Source, James Ransome and Anmol Misra, CRC Press, 2014, ISBN: 978-1466560956.

[Wiegers] Peer Reviews in Software: A Practical Guide, Karl Wiegers, Addison-Wesley Professional, 2001, ISBN: 978-0201734850

Web

[CodeProject] Code review guidelines, https://www.codeproject.com/articles/524235/codeplusreviewplusguidelines

[Howard2] Howard, M. (2004, November). “Attack Surface: Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users.”, http://download.microsoft.com/download/3/a/7/3a7fa450-1f33-41f7-9e6d-3aa95b5a6aea/MSDNMagazineNovember2004en-us.chm

[Howard3] Howard, M. (2003, November). “Review It: Expert Tips for Finding Security Defects in Your Code”, http://download.microsoft.com/download/3/a/7/3a7fa450-1f33-41f7-9e6d-3aa95b5a6aea/MSDNMagazineNovember2003en-us.chm

[Meier] Meier, J., et al. (2005, October). “How To: Perform a Security Code Review for Managed Code (.NET Framework 2.0)”. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649315(v%3dpandp.10)

[OWASP] OWASP Code Review Guide, https://www.owasp.org/images/2/2e/OWASP_Code_Review_Guide-V1_1.pdf

Research & Real World Examples

[Wojtczuk BH 2009] Attack Intel BIOS,https://www.blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf

[Rutkowska BH DC 2009] Attack Intel TXT,https://www.blackhat.com/presentations/bh-dc-09/Wojtczuk_Rutkowska/BlackHat-DC-09-Rutkowska-Attacking-Intel-TXT-slides.pdf

[Bazhaniuk CSW 2015] A New Class of Vulnerability in SMI handlers, http://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf

[ThinkPwn 2016] Exploring Lenovo,http://blog.cr4.sh/2016/06/exploring-and-exploiting-lenovo.html

[Wojtczuk 2011] Attacking Intel TXT via SINIT Hijacking, https://invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf

[Kallenberg 2014] Extreme Privilege Escalation on Windows 8 UEFI System, https://www.mitre.org/sites/default/files/publications/14-2221-extreme-escalation-presentation.pdf

[Kallenberg CSW 2014] All your boot are belong to us, https://cansecwest.com/slides/2014/AllYourBoot_csw14-mitre-final.pdf

[Wojtczuk CSW 2015] Attacks on UEFI Security, https://cansecwest.com/slides/2015/AttacksOnUEFI_Rafal.pptx

[Evdokimov BH 2017] Intel AMT Stealth Breakthrough,https://www.blackhat.com/docs/us-17/thursday/us-17-Evdokimov-Intel-AMT-Stealth-Breakthrough.pdf

[SpeedRacer 2014] Speed Racer, https://fahrplan.events.ccc.de/congress/2014/Fahrplan/system/attachments/2565/original/speed_racer_whitepaper.pdf

[Rutkowska BH 2008] Preventing and Detecting Xen Hypervisor Subversions, https://invisiblethingslab.com/resources/bh08/part2-full.pdf

[Tereshkin BH 2009] A Ring -3 Rootkits,https://invisiblethingslab.com/resources/bh09usa/Ring%20-3%20Rootkits.pdf

[Bulygin RC 2017] Baring the system, http://www.c7zero.info/stuff/REConBrussels2017_BARing_the_system.pdf

[Wojtczuk 2009] Another TXT Attack, https://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf

[Duflot CSW 2009] SMM Reloaded,https://cansecwest.com/csw09/csw09-duflot.pdf

[Sevinsky BH 2013] Funderbolt – Adventures in thunderbolt DMA attacks,https://media.blackhat.com/us-13/US-13-Sevinsky-Funderbolt-Adventures-in-Thunderbolt-DMA-Attacks-Slides.pdf

[Pcileech 2017] Attacking UEFI and Linux, http://blog.frizk.net/2017/01/attacking-uefi-and-linux.html

[Facedancer 2012] Facedancer, http://goodfet.sourceforge.net/hardware/facedancer21/

[Schumilo BH 2014] Don’t trust your USB,https://www.blackhat.com/docs/eu-14/materials/eu-14-Schumilo-Dont-Trust-Your-USB-How-To-Find-Bugs-In-USB-Device-Drivers.pdf

[Boone CSW 2018] TPM Genie, https://github.com/nccgroup/TPMGenie/blob/master/docs/CanSecWest2018-TPM_Genie-_Jeremy_Boone.pdf

[Brossard DC 2008] Bypassing Pre-boot Authentication Passwords, https://www.defcon.org/images/defcon-16/dc16-presentations/brossard/defcon-16-brossard-wp.pdf

[Miller BH 2011] Battery Firmware Hacking, https://media.blackhat.com/bh-us-11/Miller/BH_US_11_Miller_Battery_Firmware_Public_Slides.pdf

[Duflot 2006] Using CPU System Management Mode to Circumvent Operating System Security Function, https://www.researchgate.net/publication/241643659_Using_CPU_System_Management_Mode_to_Circumvent_Operating_System_Security_Functions

[CIH 1998] CIH, https://en.wikipedia.org/wiki/CIH_(computer_virus))

[Sun 2007] BIOS Boot Hijacking, http://powerofcommunity.net/poc2007/sunbing.pdf

[Ermolov 2016] Safeguarding Rootkits: Intel Boot Guard, https://github.com/flothrone/bootguard/blob/master/Intel%20BootGuard%20final.pdf

[Ermolov DC 2017] Safeguarding Rootkits: Intel Boot Guard (part2),https://github.com/flothrone/bootguard/blob/master/Intel%20BG%20part2.pdf

[Matrosov BH 2017] Betraying the BIOS, https://www.blackhat.com/docs/us-17/wednesday/us-17-Matrosov-Betraying-The-BIOS-Where-The-Guardians-Of-The-BIOS-Are-Failing.pdf

[Matrosov BH 2019] Modern Secure Boot Attacks,http://i.blackhat.com/asia-19/Fri-March-29/bh-asia-Matrosov-Modern-Secure-Boot-Attacks.pdf

[Han BH 2018] I don't want to sleep tonight – Subverting Intel TXT with S3 Sleep, https://i.blackhat.com/briefings/asia/2018/asia-18-Seunghun-I_Dont_Want_to_Sleep_Tonight_Subverting_Intel_TXT_with_S3_Sleep.pdf

[Han BH 2019] Finally I can sleep tonight – catching sleep mode vulnerabilities of the TPM with the napper, http://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Seunghun-Finally-I-Can-Sleep-Tonight-Catching-Sleep-Mode-Vulnerabilities-of-the-TPM-with-the-Napper.pdf

[Chen BH 2009] Reversing and exploiting an Apple firmware update, https://www.blackhat.com/presentations/bh-usa-09/CHEN/BHUSA09-Chen-RevAppleFirm-SLIDES.pdf

[Weinmann 2010] The hidden nemesis,https://media.ccc.de/v/27c3-4174-en-the_hidden_nemesis/related

[Cui BH 2011] Print me if you dare,https://academiccommons.columbia.edu/doi/10.7916/D8QJ7RG3

[Domas BH 2018] God Mode Unlocked Hardware Backdoors in X86 CPUs, http://i.blackhat.com/us-18/Thu-August-9/us-18-Domas-God-Mode-Unlocked-Hardware-Backdoors-In-x86-CPUs.pdf

Authors

Jiewen Yao (jiewen.yao@intel.com) is a Principal Engineer with Intel Architecture, Graphic and Software Group at Intel Corporation. He is security architect in EDK II BIOS. Jiewen is member of UEFI Security Sub-team and PI Security Sub-team in the UEFI Forum.

Chris Wu (chris.wu@intel.com) is a validation leader with Intel Architecture, Graphic and Software Group at Intel Corporation.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer with Intel Architecture, Graphic and Software Group at Intel Corporation. Vincent chairs the UEFI Security and Network Sub-teams in the UEFI Forum.