White Paper

A Tour Beyond BIOS - Security Design Guide in EDKII

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

September 2016

Executive Summary

Introduction

The purpose of this document is to provide security guidelines to developers, implementers, and code reviewers of the EDKII firmware. The topics discussed below are intended to aid in reducing bugs associated with common security vulnerability classes present in EDK II. Following these guidelines will increase the overall security of platforms implementing the firmware and ensure platforms are not as susceptible to malicious behavior.

Audience

This document is intended to be useful to those designing new contributions for tianocore.org and those contributing code and reviewing code for tianocore.org. It is also intended to be useful by those using tianocore.org (and other boot firmware) content.

Table of Contents

Executive Summary ii
Introduction ii
Audience ii
Overview 6
Document structure 6
Conventions 6
General Design Guidelines 7
Input validation 7
Defense in depth 8
Cryptography and Secrets 8
Size of security code 8
Reduce the Attack Surface 8
Economy of mechanism 9
Error recovery 9
Privileges 9
User Interface 10
Authorization 10
Consider Resiliency in Design 10
General Coding Guidelines 11
Avoid Arithmetic Errors 11
Prevent Buffer Overruns 11
Be aware of 11
Post-Coding Guidance 12
Practicality 12
Boot Firmware Specific Guidelines 13
1. Verify input from untrusted sources 13
2. Assume any code shipped to a customer will be used in a product 13
3. UEFI Variable use and misuse 13
4. Do not store secrets in the clear 14

5. Asserts are not a substitute for error recovery 14
SMM 16
6. Protect SMM from access by code and I/O outside of SMM 16
7. Data from outside SMM is not inherently trusted 16
8. Do not call outside of SMM from inside SMM. 16
9. Use existing SMM functions to transfer parameter blocks to and from SMM. 16
10. Deprecated APIs 16
11. Use existing protections where available 17
12. Avoid unnecessary privilege escalations 17
13. Setup security configuration 17
14. S3 Script 17
15. Lock critical registers in normal boot and S3 path 18
16. BIOS update 18
17. BIOS recovery 18
Appendix A – Examples 19
SMM Communication 19
PE COFF Image 20
BMP Image File 21
UEFI Variable 22
S3 Configuration Data 23
Capsule Coalesce 24
StrCpyS/StrCatS/StrnCpyS/StrnCatS 25
Appendix B – EDKII Code Review Top5 28
If the code is SMM related, ... 28
If the code touches UEFI Variables, ... 28
If the code consumes input from an untrusted source or region, ... 28
Verify buffer overflow is handled. Avoid integer overflow. 28
Verify that ASSERT is used properly. 29
Acknowledgments 30
References 31
Books and Papers 31

Web 31

Overview

Document structure

Following introductory remarks in this section, this document has three major sections:

1. General design guidelines
2. General coding guidelines
3. Specific guidelines

The first two sections are culled from industry secure coding lists with examples relevant to the users of this site. The third section arises from lessons learned over the life of UEFI / PI / ACPI / EDK II due to review, research, and a few escapes.

Conventions

"UEFI Variables" are non-volatile storage as defined in section 7 (Runtime Services) of the UEFI Specification. "Variables" are C variables such as

```
UINT32    DataLength;
```

In the following references of the form:

CWE-*xxx* (*n*/25) *Title*

Refer to items in Mitre's Common Weakness Exchange (CWE) Top 25 [Top25], with xxx being the CWE identifying one to three digit number, *n* being the ranking in the top 25 (so 3 would be the third most common issue) and *Title* being the CWE issue name. Due to applicability, less than 25 items are referenced. In most cases the remaining write-ups at [CWE] can be used as is. In a few cases, the write-ups are e.g. web-oriented but there are parallel issues in firmware. Those cases are noted following the title.

General Design Guidelines

The following list is a compilation of the lists mentioned in the references below along with specific additions and accumulated knowledge. Where available, we provide a pointer to one of the referenced lists that discusses the issue in question. The lists have considerable overlap (and were inevitably derived from one another as well) so several lists may cover the same topics. The basic structure of the list is from [Ransome, Chapter 5.4], which was derived from the earlier Saltzer and Shroeder paper from 1974 [Saltzer]. The Saltzer paper is widely viewed as the first serious attempt at such a list. This list below is reordered to be more consistent with relevance to firmware.
Due to their influence, the relevant Mitre Corp. Common Weakness Exchange (CWE) Top 25 items are included at the end of each topic for further reference.

Input validation

1. Validate all input from untrusted sources. Input from an end user, or across the network, or from a peripheral, or from a UEFI Variable should always be considered untrusted. Data that crosses a trust boundary from an untrusted source to a trusted one should also be considered untrusted.
2. Input validation can take several forms. At its most basic (and within size constraints) perform sanity checks and bounds checks on:
    a. Type
    b. Length
    c. Range
    d. Format
3. Check function return values in consumed (called or otherwise invoked) API. [Msdn]
4. Validate input as much as possible in the routine that first encounters the untrusted input. Then validate the input more as the structure of the input becomes more constrained in subsequent routines. This guideline must be balanced against coding time / performance / size constraints although most input validation requires nearly no additional time or size. There are many ways to achieve defense in depth. This is one.
5. The most common (standard) internal character encoding in EDK II is UTF-16/UCS-2 (CHAR16) but that does not mean it is universal: You will also find UTF-8 and ASCII at least. Ensure that you understand how the strings your code is processing are encoded.
6. Sanity check data from add-in hardware (e.g. SPD data from DIMMs).
7. Use canonical paths for comparisons. In practice, this usually means reformatting all input into a consistent format (say always '/' rather than '/' or '\' in file paths). This allows sanity checking during the reformatting process and makes comparisons easier later. [Msdn]

See CWE-807 (10/25) Reliance on Untrusted Inputs in a Security Decision; CWE-494 (14/25) Download of Code Without Integrity Check; CWE-306 (5/25) Missing Authentication for Critical Function

Defense in depth

8. If an attack succeeds at accessing a minor asset does it also gain access to more major assets: Does the attack lose the equivalent of "one village on the border or the whole kingdom?" [Ransome]
9. Doing the same checks on the same data several times is not the intent of (or good) defense in depth. Instead, each layer in the solution must validate the requests it is being given in light of what it knows about the current state of the system. The protections should be additive, not repetitive.
10. Are the modules defined so that it requires support of several modules to mount a successful attack? [Ransome]

Cryptography and Secrets

11. Assume all code and data is in open source. [Ransome] The security should still work if all of the source code is made public. Hiding private keys in firmware is thus not safe: Obscurity ≠ Security.
12. "Don't roll your own crypto algorithms. Always use existing optimized functions. It is very difficult to implement a secure cryptographic algorithm, and good, secure cryptographic functions are readily available." [Apple]
13. "When you store the user's password on the computer, encrypt it." [Auscert] The SHA-256 hash functions are already available in UEFI and in the security packages. Store the hash of the password (or other secret) and then zero the secret's buffer.
14. Use the existing cryptographic support. Do not invent your own. In particular, do not reinvent the random number generation routines. Call them. If you do not trust them, improve them.
15. "Scrub (zero) user passwords and other secrets from memory after validation." [Apple]
16. "Do not store unencrypted passwords." [Apple] in code or in variables or in RAM.
17. If random value is needed, try to use hardware generated random seed if supported, such as X86 RDSEED instruction. RTC is not a good random seed.

See CWE-798 (7/25) Use of Hard-coded Credentials; CWE-311 (8/25) Missing Encryption of Sensitive Data; CWE-327 (19/25) Use of a Broken or Risky Cryptographic Algorithm; CWE-759 (25/25) Use of a One-Way Hash without a Salt.

Size of security code

18. "Make the critical portion of your program as small as possible and as simple as possible." [Auscert]
19. Design functions so their results can be trusted: Design functions so that sufficient information is returned from the function to allow the caller to trust the results of the request. Return the buffer and the length of the buffer, for example.

Reduce the Attack Surface

> An interface between two programs is comprised of all of the assumptions that the programs make about each other.
>
> Nancy Leveson [Leveson]

20. Determine if you can reduce the number of attack surfaces in the design. “Attack surface reduction embodies employing layered defenses, shutting off or restricting access to system services and applying the principle of least privilege wherever possible.” ([27034 pg. 47])
21. Expose only interfaces which are intended to be used by other modules. This should be the minimum set necessary to get the job done and should be well documented to describe proper use of the interfaces and the results from invoking those interfaces.
22. Do not implement or use “undocumented” APIs. You may implement local APIs. Use only your module’s local APIs. Do not use another module’s local (also known as “private”) APIs.
23. If your project is not using a module, disable or remove it from your project. The build system makes this easy.
24. Do not use private APIs unless you are working in the module that created that API.

Economy of mechanism

25. Are the implementations of the security requirements so complex that they cannot be effectively reviewed? If so, consider rewriting them or changing the mitigation methods. [Ransome]
26. Identify and keep security critical sections few, simple, and short.

Error recovery

27. “Fail Safe: … It should be designed so that if the program does fail, the safest result should occur.” [Wheeler, 7.10]
28. Is error recovery fail safe: If there is an issue, does it open up an attack surface or (preferably) close one down? [Ransome]

Privileges

Viewed one way, very few security concerns in boot firmware are rooted in privileges and privilege management. Some examples of privilege in boot firmware include:

- Access to the code store (flash for example) could be considered a privilege.
- Certainly access to the MM (SMM, Trust Zone) is considered privileged.
- Ability to modify erase and modify important configuration options such as the UEFI secure boot variables is also viewed as requiring privileges.

29. Applications and functions should use the least privilege that will get the job done [Apple] If some work can be done in DXE, don’t put it to SMM.
30. Use privilege bracketing (do a lot of lower privilege code, go into higher privilege code for the few privileged pieces and then return to lower privilege) to enable the privilege only when necessary and then turn it off as early as possible
31. Do not disable protections to fix a bug. If you are getting a protection trap, do not turn off the protections. Instead, determine why you are triggering the trap. This can occur, for example, when attempting to call outside SMM.
32. Least common mechanism [Ransome]: If a caller is given elevated access, can another caller “piggyback” on to the increased privilege?

See CWE-250 (11/25) Execution with Unnecessary Privileges

User Interface

Boot firmware has less user interface than many other types of software so this discussion ends up lower on the list.

33. Make sure default configuration and parameters are secure settings [Apple], [Wheeler, 7.7]
34. When creating error returns or error messages, don't inadvertently assist the attacker. Consider a prompt for name and password, with error returns "Incorrect name" and "Incorrect password". Receiving "Incorrect password" means that the name is correct. Better to provide the same "Incorrect name or password" message for both (which has the added benefit of being slightly more accurate in this case).

See CWE-732 (17/25) Incorrect Permission Assignment for Critical Resource – default to protections on

Authorization

35. CWE-862 (6/25) Missing Authorization – who can run Setup? Who can modify variables?
36. CWE-307 (21/25) Improper Restriction of Excessive Authentication Attempts – protect against brute force methods for passwords, flash scrubbing.

Consider Resiliency in Design

37. Resiliency consists of 4 pillars: Protection, Detection, and Recovery, and Preparation. For a design to be truly secure, it must consider all of these.
    a. Protection: How does this code help to keep an attack from happening?
    b. Detection: How does this code help to determine that an attack has happened?
    c. Recovery: How does this code help to resume normal operation after an attack has happened?
    d. Preparation: During normal operation, how do we make it easier (or even possible) to perform the above? This might take the form of ensuring that all modules are up to date or it might be enabling methods to back up critical data (such as certain UEFI Variables) so that they can be recovered in case of a successful attack on the variable storage.

Many of the other guidelines in this list can fall into one or more of these 4 buckets.

General Coding Guidelines

These guidelines are useful when writing code and when reviewing code.

Avoid Arithmetic Errors

1. Checks based on data type limitations (e.g., integer underflows and overflows)
2. Properly cast numeric variables involved in string manipulation
3. Use unsigned values unless the contents are really signed.

See CWE-134 (23/25) Uncontrolled Format String; CWE-190 (24/25) Integer Overflow or Wraparound

Prevent Buffer Overruns

4. Check buffer sizes, copies, and indices (esp. `sizeof`) (Also see [CERT], REC.03, EXP09-C. Use sizeof to determine the size of a type or variable)
5. Check for appropriate buffer size. Maximums should be defined globally where possible to avoid assumptions in lower code layers.
6. Check for NULL Pointers dereferenced in the code.
7. NIL-terminate strings and check for NULL. Use buffer length as a first check.
8. Check both heap and stack overflows.
9. Be careful of boundary conditions (e.g., off by one errors, array indices) and conditionals (e.g., reverse logic).
10. When checking boundary conditions, try to use subtraction instead of addition, division instead of multiplication.

See CWE-131 (20/25) Incorrect Calculation of Buffer Size; CWE-120 (3/25) Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

Be aware of

13. Time of Check versus Time of Use (TOCTOU) issues [TOCTOU]: TOCTOU is a particular form of a race condition where an buffer is checked before being moved into a protected space where it is used allowing time between check and use for a third party to change the content. Consider a case where a Management Mode (MM) verifies a buffer and then reads it into MM memory. A non-rendezvoused processor could, between the two operations, write into the buffer, thus subverting the verification. TOCTOU issues can be tricky to spot but are usually fairly easy to resolve. In the example, copying the buffer into MM space and then verifying it resolves the issue (at the possible cost of efficiency and cleaner code).
14. Ensure that functions used are not deprecated. Ensure that functions defined do not deserve to be deprecated. See the deprecation discussion below.

See CWE-676 (18/25) Use of Potentially Dangerous Function

Post-Coding Guidance

15. Use security static code analyzers. [Msdn]. [StaticAnalysis] gives a list of tools for static code analysis.
16. Identify the critical paths and weakest links (if any) in the change's security? Are they sufficiently strong? [Ransome] There is a case for looking at all code as if you wished to attack that code. Playing attacker is not an easy role and requires much education and often uncomfortable changes in viewpoint to learn.
17. Perform security code reviews on security critical code and other representative sections of code. The reviewers and the mailing list should nominate changes for more intense review.
18. Inspect and validate handling of corner cases especially in the security critical code (or any code for that matter).

Practicality

You may have to prioritize opportunities for improvements including some that may have to wait until a future generation of the code. If so, you should review the prioritization with the appropriate team (the module owner and /or mailing list for open source or your security expert and possibly other experts) and record the "to do" items for future inclusion.

Boot Firmware Specific Guidelines

1. Verify input from untrusted sources

To use input from untrusted sources, you must figure out how to trust it at least to the extent you can make use of the input data. This means you first need to understand how much you need to trust the data: will you, for example, configure memory using this data, or will you simply display the information on a console (a user's name, for example). Both are untrusted input but the impact may be different.

You must also know what it takes to turn untrusted input to trusted input. At a minimum, bounds checking and sanity checking should be used. For the contents of an option ROM in a system using UEFI Secure Boot, on the other hand, the contents must be verified against the **`dbx`** and **`db`** variables.

Bounds and sanity checking include ensuring that the input does not exceed the maximum length and that the contents themselves are, if not valid semantically, are at least valid syntactically. For example, it may not be possible to verify that a device path is a valid device path on the system (semantics) but the supposed device can be sanity checked to ensure that it is correctly formed.

If you write new code that directly consumes (is the first to encounter) untrusted input, mark that code with the following statement:

```
<Caution: This function may receive untrusted input.>
```

Similarly, if you discover a piece of code that does directly consume untrusted input that is not marked, update the code with the mark. This alerts security reviewers (and others) to perform more reviews for changes.

2. Assume any code shipped to a customer will be used in a product

Experience suggests that customers will include our reference code as is, even if that code includes validation options that violate security features or code that disables security features for validation or other purposes. Defaults on reference code when shipped to customers must be secure so that the customers must proactively change code to remove product-level security. Note that product-level security may vary from product to product. Reference code targeting a builder product will have different security levels than reference code targeting a traditional PC or server. It is essential that the customers be made aware of the security constraints of the system.

3. UEFI Variable use and misuse

When you read variables (**`GetVariable`**), treat their contents as untrusted input (see above). Sanity check their contents before use. For example, assume that byte 7 of a Setup variable must be 0, 1, 2, or 5. Before using byte 7, ensure that the value is either 0, 1, 2, or 5. The entire contents of the variable may be checked at once (before the first time the variable is used) or the various components of the variable may be checked individually. Each has implications for code construction:

- Validate-All: Checking the variable content before first use may ensure that the variable is checked. It may however consume more space and be slower. As well, subsequent

code will assume that the variable contents are valid. Once the variable content is checked before first use, it is very difficult to change to validate-at-use.

- Validate-at-use: Checking the variable content at use of each component (byte, word, element in a structure, etc.) relies on each and every consumer to do their job correctly, which has proved difficult and tends to duplicate code.

Use the length from **`GetVariable`** to bounds check the variable. For null-terminated variables (such as variables containing devices paths) code must not scan beyond the returned length even if the null is missing.
All data pertaining to a variable must be stored in that variable. Do not create designs in which a single API request must update more than one variable.

4. Do not store secrets in the clear

Passwords must be stored as hashes. Use the existing hash algorithms to hash, preferably SHA2 but SHA1 and MD5 are acceptable in cases of limited storage (subject to review by the organization's security experts). XOR with 0x5A is not sufficient. Error Correcting Code (ECC) is not sufficient.
The password and other secrets should only be able to be changed or erased without authentication by using a method requiring unambiguous physical presence. That is, the design must have a method by which the code can prove that the user is physically at the system. Key presses on the keyboard are *not* sufficient since KVM boxes may be networked. The determination of unambiguous physical presence must be abstracted from its uses.

5. Asserts are not a substitute for error recovery

> Assertions are especially useful in large, complicated programs and in high-reliability programs. They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and so on.
>
> [McConnell, Chapter 8.2]

> Assertions are a shorthand way to write debugging checks. Use them to catch illegal conditions that should never arise. Don't confuse such conditions with error conditions, which you must handle in the final product.
>
> [Maguire, Chapter 2]

[McConnell] discussed several guidelines for ASSERT usage.

- Use error-handling code for conditions you expect to occur; use assertions for conditions that should never occur.
- Avoid putting executable code into assertions.
- Use assertions to document and verify preconditions and postconditions.
- For highly robust code, assert and then handle the error anyway.

These rule can be used in a UEFI BIOS as well. For the condition "be expected to occur", the BIOS must use error handling; for the condition "never occur", the BIOS can use ASSERT to make sure that.

In early SEC/PEI phase, an error might mean a system configuration error or a hardware error. It might be unrecoverable and should never occur. For example, the memory initialization module fails to find memory DIMM. In that case, the best way is to use PI ReportStatusCode to inform end user via the LED or BEEP code. ASSERT() can be used here to let developer check if memory initialization module is using wrong parameters.
However, in UEFI shell environment, it received external input from the end user, such as to list a non-existing file, or to copy file to a wrong place. These should be handled by the error handler, instead of ASSERT(). The network driver receives the network packets from the internet. It cannot assume the packet is always right one. The network driver should not use ASSERT to handle the error packet. The proper way is to check the packets and drop mal-format ones.

Below is some other examples on ASSERT usage.
For UEFI Variable API,

- GetVariable (NV+RT without AU/RO) status check should NOT ASSERT(), or at least has error handling code followed by.
- GetVariable (AU/RO) status check may ASSERT(), if the driver assumes variable must exist.
- SetVariable (NV) status check should NOT ASSERT(), or at least has error handling code followed by.
- SetVariable (without NV) status check may ASSERT(), before EndOfDxe.

For resource allocation,

- Memory Allocation status check should NOT ASSERT() after EndOfDxe.
- Memory Allocation status check may ASSERT() before EndOfDxe, if the allocation failure prevents system from booting.
- MMIO/IO Allocation status check for external device should NOT ASSERT().
- MMIO/IO Allocation status check for onboard device may ASSERT() before EndOfDxe

For SMM,

- SMI handler should NOT ASSERT() for the external input check, after EndOfDxe.
- SMM driver may ASSERT() in the entrypoint to construct the environment.

ASSERT is also used in some EDKII library functions, such as BaseLib, BaseMemoryLib, IoLib, PciLib, and HobLib. Once you've written a function, review it and ask yourself, "What am I assuming?" If you find an assumption, either assert that your assumption is always valid, or rewrite the code to remove the assumption. [Maguire]

SMM

6. Protect SMM from access by code and I/O outside of SMM

SMM must protect itself. Code must use hardware protections to protect SMM's memory from access by non-SMM code including bus mastering access.

7. Data from outside SMM is not inherently trusted

Data from outside of SMM should be treated as untrusted and, as such, meet the requirements in "Verify input from untrusted sources" above.

8. Do not call outside of SMM from inside SMM.

SMM is a mode. Once in SMM, the only way to leave SMM is via an RSM. Even if the processor executes outside of the memory reserved for SMM, it is still in SMM, although memory outside of SMM should not be trusted by SMM. Calling outside SMM requires execution of code that is untrusted, which is an extreme failure of doing "Verify input from untrusted sources" above. In a UEFI/PI BIOS, the SMM moduole should not call any UEFI boot services, runtime services, protocol interface, or dynamic PCD after SmmReadyToLock.

9. Use existing SMM functions to transfer parameter blocks to and from SMM.

This enforces the rule: Do not create or update SMM so that callers outside SMM can, by calling SMM, gain access to memory that they otherwise would not have access to. (See also "Avoid unnecessary privilege escalations" below.)

10. Deprecated APIs

New code or updates to existing code should not use any deprecated interface. (Deprecated interfaces are annotated in the file headers and are protected with build error generating macros.)

1. Interfaces that an industry forum owns and has been deprecated in a published specification. UEFI deprecated some interfaces, such as Device IO protocol, UGA_IO/UGA_DRAW protocol, USB Host Controller Protocol, SCSI PassThru protocol, etc. TCG defines TCG2 protocol for TPM2 to deprecate TrEE protocol.
2. A few interfaces (two, both called **`GetVariable`**) have been designated as "inherently dangerous functions". The classic non-firmware example of an "inherently dangerous function" is the C library function:

   ```
   gets (s);
   ```

   since the input length cannot be limited to the size of the buffer (the size of **`s`** in the above example). [CWE 242]
3. There are also functions which some parts of the community have decided to be dangerous enough to deprecate even though they may be used safely. (See e.g. [Apple], [CWE 676], [Auscert #5])
   a. The main example of this class of functions is the '**`Str`**' functions (the equivalent of the C **`strcpy`**, **`strcat`**, etc. functions). Use e.g. **`StrCpyS`** and **`StrCatS`** instead.
   b. Additional impetus behind the effort to deprecate these functions arises because many the SDLs (Secure Development Life cycles) and static analysis tools require these "banned" functions to be removed.

c. We have not changed the interfaces to `MemCpy` because, while there are a lot of instances of `MemCpy` in the existing codebase, we have not seen many, if any, issues related to `MemCpy`.
d. We do not require third party code (e.g OpenSSL) to be modified to meet this requirement.

11. Use existing protections where available

If we can use the same hardware the Operating System will rely on (the processor's page table access control bits, for example) to perform protection, we do not need to invent new protections. We must use alternative protections in cases where the OS will reconfigure, or even has access to, the hardware's protections as part of its normal operation. So, for example, the page tables can be used in SMM (where the firmware 'owns' the page tables) but not in normal execution, where the OS owns the page tables after it is loaded.

12. Avoid unnecessary privilege escalations

Arguably the firmware at the reset vector has more privileges than any other code that executes in the system (or at least the main processor). One of the methods by which the system becomes more secure is to turn off some of the privileges the system booted with.
SMM has run-time privileges that Operating Systems and Hypervisors go to great lengths to ensure that most code does not have. SMM must not provide a means of allowing lower privilege application code to bypass Operating System or Hypervisor protections. This can occur in a number of ways. For example, if a synchronous SMI requires a buffer and the pointer to that buffer is passed in, SMM has no good way to verify that the pointer points to a space that the caller has access to. The preferred mitigation is to pre-allocate a buffer prior to OS load which can form a mailbox between the OS and SMM. The OS can then control access to the buffer, which should be allocated in firmware reserved RAM. The SMI handler need verify the source of communication buffer and block access the memory other than the pre-allocated reserved buffer.

13. Setup security configuration

Do not provide setup options to enable or disable flash protections on code. Code should be and can be protected better than setup variables. By adding configuration for flash protection features, we make the protection of the flash code no better than protection of setup variables.
Proof of physical presence is a requirement that the user prove to the firmware that the user is actually at the system being configured, rather than communicating remotely. Proof of "Physical presence" is required during the provisioning and maintenance phases of some security standards, including those related to TPM. Simply running Setup is not sufficient to prove "physical presence" since the keyboard and video which control setup may be proxied remotely via a KVM.

14. S3 Script

The S3 script must be stored in a secure space and / or verified before use during S3 resume. "Secure space" can be SMM but cannot be any space that the OS has write access to. The existing SMM "Lock Box" should be used by default.
If you choose not to use SMM to protect S3 script, the following is an alternative. At cold boot, store the script in OS accessible reserved RAM at boot. Then hash (not checksum, not error

correcting code, but SHA-256 hash) the script and store the hash in e.g. SMM or some other protected space. Any space accessible to the OS is insufficiently protected. Variables are not preferred but are allowed. If you are using a variable, the variable must be non-volatile (since volatile variables are not well enough protected). The variables must also be protected from write by the OS.
Upon S3 resume *AND* before using the script, hash the script as it exists in memory and compare with the hash calculated at cold boot. If the two are not equal, treat the reboot as a cold boot.
S3 script Execute operations (which call code) must only invoke code that is trusted. Simply storing code in BIOS reserved memory is not sufficient. Instead, trust can be maintained by one of several methods:

- Through protection of the code itself, say by running execute-in-place code out of flash or by running code from inside SMM.
- By hashing the code and storing the hash in a protected place at cold boot, and then verifying the hash during S3 resume *before the code is invoked*. If the hash fails, the S3 resume has failed and the system should be forced through a cold boot cycle.
- By saving the code in Lock Box and reloading the code to the same locations during each S3 resume cycle.

15. Lock critical registers in normal boot and S3 path

If a silicon provides the “lock” capability for some critical registers, including but not limited to SMRAM, flash part, and MMIO configuration BAR, they must be locked before EndOfDxe. These “lock” actions must also be performance in S3 resume path. These actions can be registered in the boot script or be in a PEIM.

16. BIOS update

If a BIOS provides flash update capability, the update code must be in an isolated execution environment, such as SMM, or before EndOfDxe. The updatable image must be signed and have the version information. The update code must reject any unsigned image and verify the signature. The update code must check the new version in the updatable image and make sure the new version is greater than or equal to the lowest supported version of the current image.

17. BIOS recovery

If a BIOS provides recovery capability and the recovery image is from an external source such as USB key, the recovery module must be in firmware boot block. The recovery image must be signed and have the version information. The recovery module must reject any unsinged image and verify the signature. The recovery module must check the version in recovery image and make sure the recovery module version is same as the current image. If the recovery image is from an internal source, such as another flash part or chip, the recovery module may skip the signature verification and version check. The BIOS need adopt same flash update protection to protect the recovery module on flash.

Appendix A – Examples

Below are some examples of real issue, they are fixed in EDKII. It is not the full list of security issues, but a list of typical examples to show how we should design and code the UEFI BIOS.
We use [GDG-x] for "General Design Guideline", [GCG-x] for "General Coding Guideline", and [BFSG-x] for "Boot Firmware Specific Guideline" as reference. Last x means the index of guideline.

SMM Communication

[GDG-1], [GCG-13], [BFSG-1], [BFSG-7]
When a DXE agent or OS agent uses SMM communication buffer talk with SMM handler, the SMM handler must validate the source before use it. The reason is that the SMM communication buffer might be constructed by malware. Care must be taken to avoid TOC-TOU issue.

MdeModulePkg\Universal\Variable\RuntimeDxe\VariableSmm.c

```
======= old Version 9d00d20ed40fb56d8b5a8e1a3f7ae3e491ceaf94 ============================
    case SMM_VARIABLE_FUNCTION_GET_VARIABLE:
      SmmVariableHeader = (SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE *)
SmmVariableFunctionHeader->Data;
      if (((UINTN)(~0) - SmmVariableHeader->DataSize <
OFFSET_OF(SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE, Name)) ||
          ((UINTN)(~0) - SmmVariableHeader->NameSize <
OFFSET_OF(SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE, Name) + SmmVariableHeader-
>DataSize)) {
        //
        // Prevent InfoSize overflow happen
        //
        Status = EFI_ACCESS_DENIED;
        goto EXIT;
      }
=================================
======= New Version 5e5bb2a9baefcd2f231696ea94576dab5565fbfb ============================
    case SMM_VARIABLE_FUNCTION_GET_VARIABLE:
      if (CommBufferPayloadSize < OFFSET_OF(SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE,
Name)) {
        DEBUG ((EFI_D_ERROR, "GetVariable: SMM communication buffer size
invalid!\n"));
        return EFI_SUCCESS;
      }
      //
      // Copy the input communicate buffer payload to pre-allocated SMM variable
buffer payload.
      //
      CopyMem (mVariableBufferPayload, SmmVariableFunctionHeader->Data,
CommBufferPayloadSize);
      SmmVariableHeader = (SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE *)
mVariableBufferPayload;
      if (((UINTN)(~0) - SmmVariableHeader->DataSize <
OFFSET_OF(SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE, Name)) ||
          ((UINTN)(~0) - SmmVariableHeader->NameSize <
OFFSET_OF(SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE, Name) + SmmVariableHeader-
>DataSize)) {
        //
        // Prevent InfoSize overflow happen
        //
        Status = EFI_ACCESS_DENIED;
        goto EXIT;
      }
```

```
==================================
```

PE COFF Image

[GDG-1], [GCG-1], [BFSG-1]

The PE/COFF image is considered as external input from Option ROM or OS loader in disk.

For secure boot, the BIOS need validate the PE/COFF image signature, so we must validate the PE/COFF format before parse it, to reject mal-format PE/COFF image.

MdePkg\Library\BasePeCoffLib\BasePeCoff.c

======= old Version `be30ddf795e575696d8da0636bac8c252355a1f1` ==========

```
    if (Magic == EFI_IMAGE_NT_OPTIONAL_HDR32_MAGIC) {
      //
      // Use PE32 offset
      //
      ImageContext->ImageType         = Hdr.Pe32->OptionalHeader.Subsystem;
      ImageContext->ImageSize         = (UINT64)Hdr.Pe32->OptionalHeader.SizeOfImage;
      ImageContext->SectionAlignment  = Hdr.Pe32->OptionalHeader.SectionAlignment;
      ImageContext->SizeOfHeaders     = Hdr.Pe32->OptionalHeader.SizeOfHeaders;
==================================
```

======= new Version `28186d45660c92b8d98b8b19b5f8e6ff71ea5fba` ==========

```
    if (Magic == EFI_IMAGE_NT_OPTIONAL_HDR32_MAGIC) {
      //
      // 1. Check FileHeader.SizeOfOptionalHeader filed.
      //
      if (EFI_IMAGE_NUMBER_OF_DIRECTORY_ENTRIES < Hdr.Pe32->OptionalHeader.NumberOfRvaAndSizes) {
        return RETURN_UNSUPPORTED;
      }

      if (Hdr.Pe32->FileHeader.SizeOfOptionalHeader != sizeof (EFI_IMAGE_OPTIONAL_HEADER32) - (EFI_IMAGE_NUMBER_OF_DIRECTORY_ENTRIES - Hdr.Pe32->OptionalHeader.NumberOfRvaAndSizes) * sizeof (EFI_IMAGE_DATA_DIRECTORY)) {
        return RETURN_UNSUPPORTED;
      }

      //
      // 2. Check the OptionalHeader.SizeOfHeaders field.
      // This field will be use like the following mode, so just compare the result.
      // The DataDirectory array begin with 1, not 0, so here use < to compare not <=.
      //
      if (EFI_IMAGE_DIRECTORY_ENTRY_SECURITY + 1 < Hdr.Pe32->OptionalHeader.NumberOfRvaAndSizes) {
        if (Hdr.Pe32->OptionalHeader.SizeOfHeaders < (UINT32)((UINT8 *)(&Hdr.Pe32->OptionalHeader.DataDirectory[EFI_IMAGE_DIRECTORY_ENTRY_SECURITY + 1]) - (UINT8 *)&Hdr)) {
          return RETURN_UNSUPPORTED;
        }
      }

      //
      // Read Hdr.Pe32.OptionalHeader.SizeOfHeaders data from file
      //
      Size = 1;
      Status = ImageContext->ImageRead (
                               ImageContext->Handle,
                               Hdr.Pe32->OptionalHeader.SizeOfHeaders - 1,
                               &Size,
                               &BufferData
                               );
```

```
      if (RETURN_ERROR (Status)) {
        return Status;
      }

      //
      // Check the EFI_IMAGE_DIRECTORY_ENTRY_SECURITY data.
      // Read the last byte to make sure the data is in the image region.
      // The DataDirectory array begin with 1, not 0, so here use < to compare not <=.
      //
      if (EFI_IMAGE_DIRECTORY_ENTRY_SECURITY < Hdr.Pe32-
>OptionalHeader.NumberOfRvaAndSizes) {
        if (Hdr.Pe32-
>OptionalHeader.DataDirectory[EFI_IMAGE_DIRECTORY_ENTRY_SECURITY].Size != 0) {
          //
          // Check the member data to avoid overflow.
          //
          if ((UINT32) (~0) - Hdr.Pe32-
>OptionalHeader.DataDirectory[EFI_IMAGE_DIRECTORY_ENTRY_SECURITY].VirtualAddress <
              Hdr.Pe32-
>OptionalHeader.DataDirectory[EFI_IMAGE_DIRECTORY_ENTRY_SECURITY].Size) {
            return RETURN_INVALID_PARAMETER;
          }

          //
          // Read section header from file
          //
          Size = 1;
          Status = ImageContext->ImageRead (
                                     ImageContext->Handle,
                                     Hdr.Pe32-
>OptionalHeader.DataDirectory[EFI_IMAGE_DIRECTORY_ENTRY_SECURITY].VirtualAddress +
                                      Hdr.Pe32-
>OptionalHeader.DataDirectory[EFI_IMAGE_DIRECTORY_ENTRY_SECURITY].Size - 1,
                                     &Size,
                                     &BufferData
                                     );
          if (RETURN_ERROR (Status)) {
            return Status;
          }
        }
      }

      //
      // Use PE32 offset
      //
      ImageContext->ImageType         = Hdr.Pe32->OptionalHeader.Subsystem;
      ImageContext->ImageSize         = (UINT64)Hdr.Pe32->OptionalHeader.SizeOfImage;
      ImageContext->SectionAlignment  = Hdr.Pe32->OptionalHeader.SectionAlignment;
      ImageContext->SizeOfHeaders     = Hdr.Pe32->OptionalHeader.SizeOfHeaders;
==================================
```

BMP Image File

[GCG-1]

BMP file is external input. When we check format, we need avoid integer overflow.

IntelFrameworkModulePkg\Library\GenericBdsLib\BdsConsole.c

```
======= old Version 24cdd14e81bc867dfc0ed05fd6d22d4a49858adb ==========================
  BltBufferSize = BmpHeader->PixelWidth * BmpHeader->PixelHeight * sizeof
(EFI_GRAPHICS_OUTPUT_BLT_PIXEL);
  if (BltBufferSize >= SIZE_4GB) {
    //
```

```
    // If the BMP resolution is too large
    //
    return EFI_UNSUPPORTED;
  }
==================================
======= new Version d2eec3191275e0f799d42e179fc8d8a04290992f ==========================
  BltBufferSize = MultU64x32 ((UINT64) BmpHeader->PixelWidth, BmpHeader->PixelHeight);
  //
  // Ensure the BltBufferSize * sizeof (EFI_GRAPHICS_OUTPUT_BLT_PIXEL) doesn't
overflow
  //
  if (BltBufferSize > DivU64x32 ((UINTN) ~0, sizeof (EFI_GRAPHICS_OUTPUT_BLT_PIXEL)))
{
      return EFI_UNSUPPORTED;
   }
  BltBufferSize = MultU64x32 (BltBufferSize, sizeof (EFI_GRAPHICS_OUTPUT_BLT_PIXEL));
==================================
```

UEFI Variable

[GDG-20], [BFSG-3]

Many times, a UEFI variable is used to store module level information. When we set attribute, we need ask: if NV must be added? If RT must be added? NV+RT means an attack surface for malware in OS. Or we need set RO for a variable that must be read only for OS.

IntelFrameworkModulePkg\Universal\Acpi\AcpiS3SaveDxe\AcpiVariableThunkPlatform.c

```
======= old Version 73f0127f98adec4beb5f235face3a515e380cfb7 ==========================
    mAcpiVariableSetCompatibility = AllocateMemoryBelow4G (EfiACPIMemoryNVS,
sizeof(ACPI_VARIABLE_SET_COMPATIBILITY));
    Status = gRT->SetVariable (
                    ACPI_GLOBAL_VARIABLE,
                    &gEfiAcpiVariableCompatiblityGuid,
                    EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS |
EFI_VARIABLE_NON_VOLATILE,
                    sizeof(mAcpiVariableSetCompatibility),
                    &mAcpiVariableSetCompatibility
                    );
    ASSERT_EFI_ERROR (Status);
==================================
======= new Version ef4defca7a2b8b3bab11c51e92c7a82f9ab1de84 ==========================
    mAcpiVariableSetCompatibility = AllocateMemoryBelow4G (EfiACPIMemoryNVS,
sizeof(ACPI_VARIABLE_SET_COMPATIBILITY));
    Status = gRT->SetVariable (
                    ACPI_GLOBAL_VARIABLE,
                    &gEfiAcpiVariableCompatiblityGuid,
                    EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_NON_VOLATILE,
                    sizeof(mAcpiVariableSetCompatibility),
                    &mAcpiVariableSetCompatibility
                    );
    if (!EFI_ERROR (Status)) {
      //
      // Register callback function upon VariableLockProtocol
      // to lock ACPI_GLOBAL_VARIABLE variable to avoid malicious code to update it.
      //
      EfiCreateProtocolNotifyEvent (
        &gEdkiiVariableLockProtocolGuid,
        TPL_CALLBACK,
        VariableLockAcpiGlobalVariable,
        NULL,
        &Registration
        );
```

```
    } else {
      DEBUG ((EFI_D_ERROR, "FATAL ERROR: AcpiVariableSetCompatibility cannot be saved:
%r. OS S3 may fail!\n", Status));
      gBS->FreePages (
             (EFI_PHYSICAL_ADDRESS) (UINTN) mAcpiVariableSetCompatibility,
             EFI_SIZE_TO_PAGES (sizeof (ACPI_VARIABLE_SET_COMPATIBILITY))
             );
      mAcpiVariableSetCompatibility = NULL;
    }
  }
==================================
```

S3 Configuration Data

[BFSG-14]
All S3 data must be saved in secure place, so we introduced LockBox concept.

MdeModulePkg\Library\PiDxeS3BootScriptLib\BootScriptSave.c
======= new Version ==========================

```
/**
  This function save boot script data to LockBox.

**/
VOID
SaveBootScriptDataToLockBox (
  VOID
  )
{
  EFI_STATUS            Status;

  //
  // Save whole memory copy into LockBox.
  // It will be used to restore data at S3 resume.
  //
  Status = SaveLockBox (
             &mBootScriptDataGuid,
             (VOID *)mS3BootScriptTablePtr->TableBase,
             EFI_PAGES_TO_SIZE (mS3BootScriptTablePtr->TableMemoryPageNumber)
             );
  ASSERT_EFI_ERROR (Status);

  Status = SetLockBoxAttributes (&mBootScriptDataGuid,
LOCK_BOX_ATTRIBUTE_RESTORE_IN_PLACE);
  ASSERT_EFI_ERROR (Status);

  //
  // Just need save TableBase.
  // Do not update other field because they will NOT be used in S3.
  //
  Status = SaveLockBox (
             &mBootScriptTableBaseGuid,
             (VOID *)&mS3BootScriptTablePtr->TableBase,
             sizeof(mS3BootScriptTablePtr->TableBase)
             );
  ASSERT_EFI_ERROR (Status);

  Status = SetLockBoxAttributes (&mBootScriptTableBaseGuid,
LOCK_BOX_ATTRIBUTE_RESTORE_IN_PLACE);
  ASSERT_EFI_ERROR (Status);
}
==================================
```

Capsule Coalesce

[GDG-1], [GCG-1], [BFSG-1]

Capsule BIOS update image is parsed from OS runtime. But it might be constructed by malware.

MdeModulePkg\Universal\CapsulePei\Common\CapsuleCoalesce.c

```
======= old Version dc204d5a0fd64d1ccbc90ebea827e7ad73b71f4d ============================
  while ((Ptr->Length != 0) || (Ptr->Union.ContinuationPointer !=
(EFI_PHYSICAL_ADDRESS) (UINTN) NULL)) {
    //
    // Make sure the descriptor is aligned at UINT64 in memory
    //
    if ((UINTN) Ptr & 0x07) {
      DEBUG ((EFI_D_ERROR, "BlockList address failed alignment check\n"));
      return NULL;
    }

    if (Ptr->Length == 0) {
      //
      // Descriptor points to another list of block descriptors somewhere
      // else.
      //
      Ptr = (EFI_CAPSULE_BLOCK_DESCRIPTOR  *) (UINTN) Ptr->Union.ContinuationPointer;
    } else {
      //
      //To enhance the reliability of check-up, the first capsule's header is checked
here.
      //More reliabilities check-up will do later.
      //
==================================
======= new Version ff284c56a11a9a9b32777c91bc069093d5b5d8a9 ============================
  while ((Ptr->Length != 0) || (Ptr->Union.ContinuationPointer !=
(EFI_PHYSICAL_ADDRESS) (UINTN) NULL)) {
    //
    // Make sure the descriptor is aligned at UINT64 in memory
    //
    if ((UINTN) Ptr & (sizeof(UINT64) - 1)) {
      DEBUG ((EFI_D_ERROR, "ERROR: BlockList address failed alignment check\n"));
      return NULL;
    }
    //
    // Sanity Check
    //
    if (Ptr->Length > MAX_ADDRESS) {
      DEBUG ((EFI_D_ERROR, "ERROR: Ptr->Length(0x%lx) > MAX_ADDRESS\n", Ptr->Length));
      return NULL;
    }

    if (Ptr->Length == 0) {
      //
      // Sanity Check
      //
      if (Ptr->Union.ContinuationPointer > MAX_ADDRESS) {
        DEBUG ((EFI_D_ERROR, "ERROR: Ptr->Union.ContinuationPointer(0x%lx) >
MAX_ADDRESS\n", Ptr->Union.ContinuationPointer));
        return NULL;
      }
      //
      // Descriptor points to another list of block descriptors somewhere
      // else.
      //
```

```
      Ptr = (EFI_CAPSULE_BLOCK_DESCRIPTOR  *) (UINTN) Ptr->Union.ContinuationPointer;
      DEBUG ((EFI_D_INFO, "Ptr(C) - 0x%x\n", Ptr));
      DEBUG ((EFI_D_INFO, "Ptr->Length - 0x%x\n", Ptr->Length));
      DEBUG ((EFI_D_INFO, "Ptr->Union - 0x%x\n", Ptr->Union.ContinuationPointer));
    } else {
      //
      // Sanity Check
      //
      if (Ptr->Union.DataBlock > (MAX_ADDRESS - (UINTN)Ptr->Length)) {
        DEBUG ((EFI_D_ERROR, "ERROR: Ptr->Union.DataBlock(0x%lx) > (MAX_ADDRESS -
(UINTN)Ptr->Length(0x%lx))\n", Ptr->Union.DataBlock, Ptr->Length));
        return NULL;
      }

      //
      //To enhance the reliability of check-up, the first capsule's header is checked
here.
      //More reliabilities check-up will do later.
      //
==================================
```

StrCpyS/StrCatS/StrnCpyS/StrnCatS

[BFSG-10]

We notice that old StrCpy/StrCat/StrnCat/StrnCat are marked as bad API, so we introduced safe C library version of StrCpyS/StrCatS/StrnCpyS/StrnCatS. They are similar as C11 standard. The only difference is that when error happens, EDKII version does not modify original source and return error directly.

MdePkg\Include\Library\BaseLib.h

```
==================================
/**
  Copies the string pointed to by Source (including the terminating null char)
  to the array pointed to by Destination.

  If Destination is not aligned on a 16-bit boundary, then ASSERT().
  If Source is not aligned on a 16-bit boundary, then ASSERT().
  If an error would be returned, then the function will also ASSERT().

  @param  Destination              A pointer to a Null-terminated Unicode string.
  @param  DestMax                  The maximum number of Destination Unicode
                                   char, including terminating null char.
  @param  Source                   A pointer to a Null-terminated Unicode string.

  @retval RETURN_SUCCESS           String is copied.
  @retval RETURN_BUFFER_TOO_SMALL  If DestMax is NOT greater than StrLen(Source).
  @retval RETURN_INVALID_PARAMETER If Destination is NULL.
                                   If Source is NULL.
                                   If PcdMaximumUnicodeStringLength is not zero,
                                    and DestMax is greater than
                                    PcdMaximumUnicodeStringLength.
                                   If DestMax is 0.
  @retval RETURN_ACCESS_DENIED     If Source and Destination overlap.
**/
RETURN_STATUS
EFIAPI
StrCpyS (
  OUT CHAR16       *Destination,
  IN  UINTN        DestMax,
  IN  CONST CHAR16 *Source
```

```
  );

/**
  Copies not more than Length successive char from the string pointed to by
  Source to the array pointed to by Destination. If no null char is copied from
  Source, then Destination[Length] is always set to null.

  If Length > 0 and Destination is not aligned on a 16-bit boundary, then ASSERT().
  If Length > 0 and Source is not aligned on a 16-bit boundary, then ASSERT().
  If an error would be returned, then the function will also ASSERT().

  @param  Destination              A pointer to a Null-terminated Unicode string.
  @param  DestMax                  The maximum number of Destination Unicode
                                   char, including terminating null char.
  @param  Source                   A pointer to a Null-terminated Unicode string.
  @param  Length                   The maximum number of Unicode characters to copy.

  @retval RETURN_SUCCESS           String is copied.
  @retval RETURN_BUFFER_TOO_SMALL  If DestMax is NOT greater than
                                   MIN(StrLen(Source), Length).
  @retval RETURN_INVALID_PARAMETER If Destination is NULL.
                                   If Source is NULL.
                                   If PcdMaximumUnicodeStringLength is not zero,
                                    and DestMax is greater than
                                    PcdMaximumUnicodeStringLength.
                                   If DestMax is 0.
  @retval RETURN_ACCESS_DENIED     If Source and Destination overlap.
**/
RETURN_STATUS
EFIAPI
StrnCpyS (
  OUT CHAR16       *Destination,
  IN  UINTN        DestMax,
  IN  CONST CHAR16 *Source,
  IN  UINTN        Length
  );

/**
  Appends a copy of the string pointed to by Source (including the terminating
  null char) to the end of the string pointed to by Destination.

  If Destination is not aligned on a 16-bit boundary, then ASSERT().
  If Source is not aligned on a 16-bit boundary, then ASSERT().
  If an error would be returned, then the function will also ASSERT().

  @param  Destination              A pointer to a Null-terminated Unicode string.
  @param  DestMax                  The maximum number of Destination Unicode
                                   char, including terminating null char.
  @param  Source                   A pointer to a Null-terminated Unicode string.

  @retval RETURN_SUCCESS           String is appended.
  @retval RETURN_BAD_BUFFER_SIZE   If DestMax is NOT greater than
                                   StrLen(Destination).
  @retval RETURN_BUFFER_TOO_SMALL  If (DestMax - StrLen(Destination)) is NOT
                                   greater than StrLen(Source).
  @retval RETURN_INVALID_PARAMETER If Destination is NULL.
                                   If Source is NULL.
                                   If PcdMaximumUnicodeStringLength is not zero,
```

```
                                    and DestMax is greater than
                                    PcdMaximumUnicodeStringLength.
                                   If DestMax is 0.
  @retval RETURN_ACCESS_DENIED     If Source and Destination overlap.
**/
RETURN_STATUS
EFIAPI
StrCatS (
  IN OUT CHAR16       *Destination,
  IN     UINTN        DestMax,
  IN     CONST CHAR16 *Source
  );

/**
  Appends not more than Length successive char from the string pointed to by
  Source to the end of the string pointed to by Destination. If no null char is
  copied from Source, then Destination[StrLen(Destination) + Length] is always
  set to null.

  If Destination is not aligned on a 16-bit boundary, then ASSERT().
  If Source is not aligned on a 16-bit boundary, then ASSERT().
  If an error would be returned, then the function will also ASSERT().

  @param  Destination              A pointer to a Null-terminated Unicode string.
  @param  DestMax                  The maximum number of Destination Unicode
                                   char, including terminating null char.
  @param  Source                   A pointer to a Null-terminated Unicode string.
  @param  Length                   The maximum number of Unicode characters to copy.

  @retval RETURN_SUCCESS           String is appended.
  @retval RETURN_BAD_BUFFER_SIZE   If DestMax is NOT greater than
                                   StrLen(Destination).
  @retval RETURN_BUFFER_TOO_SMALL  If (DestMax - StrLen(Destination)) is NOT
                                   greater than MIN(StrLen(Source), Length).
  @retval RETURN_INVALID_PARAMETER If Destination is NULL.
                                   If Source is NULL.
                                   If PcdMaximumUnicodeStringLength is not zero,
                                    and DestMax is greater than
                                    PcdMaximumUnicodeStringLength.
                                   If DestMax is 0.
  @retval RETURN_ACCESS_DENIED     If Source and Destination overlap.
**/
RETURN_STATUS
EFIAPI
StrnCatS (
  IN OUT CHAR16       *Destination,
  IN     UINTN        DestMax,
  IN     CONST CHAR16 *Source,
  IN     UINTN        Length
  );
==================================
```

Appendix B – EDKII Code Review Top5

The following are 5 issues that all code reviewers should have in mind as they review changes.

If the code is SMM related,

1) Determine if the code can be implemented outside SMM. If so, implement outside of SMM.
2) Check if there are calls from inside SMRAM to outside SMM. This is never allowed after SmmReadyToLock SMM code should not call any UEFI BootServices, UEFI RuntimeServices, UEFI protocol after SmmReadyToLock. If some data need to be accessed, SMM code should call the API before or at SmmReadyToLock and cache the data inside of SMRAM.
3) Check if outside SMRAM data accesses are correct. SMM APIs should never allow the caller to modify memory that the caller cannot modify itself, including memory inside SMM and memory that may be in e.g. a different virtual machine. Where possible, use a pre-allocated buffer in ACPI BIOS reserved memory.
4) If PCD is used in SMM, make sure SMM code will not call PcdGet/PcdSet for dynamic PCD after EndOfDxe

If the code touches UEFI Variables,

Ensure that the RT, NV, and RO flags are used properly. Per UEFI specification, RT is used only if the variable need runtime access and NV is used only if the variable need to be persistent cross reset. If a variable need to be ReadOnly after EndOfDxe, please lock it before EndOfDxe. For example, use `EDKII_VARIABLE_LOCK_PROTOCOL`.

If the code consumes input from an untrusted source or region,

Ensure that the input is rigorously and adequately validated. Treat the contents of UEFI Variables as untrusted input.

Verify buffer overflow is handled. Avoid integer overflow.

Try to use subtraction instead of addition and division instead of multiplication. (Example below from [Apple])

Bad

```
size_t bytes = n * m;
if (bytes < n || bytes < m) { /* BAD BAD BAD */
... /* allocate “bytes” space */
}
```

Good

```
if (n > 0 && m > 0 && SIZE_MAX/n >= m) {
size_t bytes = n * m;
... /* allocate “bytes” space */
}
```

Verify that ASSERT is used properly.

ASSERT is used to catch conditions that should *never* happen. If something *might* happen, use error handling instead. We can use both ASSERT and error handling to facilitate debugging – ASSERT allows for earlier detection and isolation of several classes of issues. [McConnell]

Acknowledgments

We would like to thank Robert Hale who contributed the initial version of this document. We would like to thank Lelia Barlow, Erik Bjorge, Mark Doran, Lee Rosenbaum, and Topher Timzen who reviewed this document and gave us valuable feedback.

References

Books and Papers

[27034] *ISO/IEC 27034-1 Information technology – Security techniques – Application Security – Part 1: Overview and concepts*, ISO/IEC 27034-1:2011(E). Particularly Annex 1.

[Graff] *Secure Coding: Principles & Practices*, M.G. Graff and K.R. van Wyk, O'Reilly, 2002, ISBN: 978-0596002428

[HowardLeBlanc] *Writing Secure Code: Practical Strategies and Proven Techniques for Building Secure Applications in a Networked World, Second Edition*, Michael Howard, David LeBlanc, Microsoft, 2004, ISBN: 978-0735617223

[Howard] *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*, Michael Howard, David LeBlanc, John Viega, McGraw-Hill, 2009, ISBN: 978-0071626750

[McConnell] *Code Complete: A Practical Handbook of Software Construction, Second Edition*, Steve McConnell, Microsoft, 2004, ISBN: 978-0735619678

[Maguire] *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs*, Steve Maguire, Microsoft, 1993, ISBN: 978-1556155512

[Ransome] *Core Software Security: Security at the Source*, James Ransome and Anmol Misra, CRC Press, 2014, ISBN: 978-1466560956. Particularly, chapters 5 and 9.

[Leveson] *Safeware: System Safety and Computers*, Nancy V. Leveson, Addison Wesley, 1995, ISBN: 978-0201119725.

[Saltzer] The Protection of Information in Computer Systems, CACM 17,7 (July 1974) available at http://www.cs.virginia.edu/~evans/cs551/saltzer/

[Teer] *Solaris Systems Programming*, Chapter 9, Secure C Programming, Rich Teer, Prentice Hall, 2007, ISBN: 978-0768682236

[Viega] *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More*. John Viega, Matt Messier, O'Reilly Media, 2003, ISBN: 978-0596003944

[ViegaMcGraw] *Building Secure Software: How to Avoid Security Problems the Right Way*, John Viega, Gary McGraw, Addison-Wesley Professional, 2001, ISBN: 978-0201721522

Web

[Apple] "Secure Coding Guide", https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Introduction.html

[Auscert] "Secure Unix Programming Checklist", https://www.auscert.org.au/render.html?it=1975

[CERT] "CERT C Coding Standard", https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard

[CWE] "Common Weakness Exchange", https://cwe.mitre.org/

[Jordan] “Ten dos and don’ts for secure coding”, Michael Jordan, http://searchsecurity.techtarget.com/searchSecurity/downloads/Jordan_Checklist.pdf
[Linux] “Secure Programming for Linux and Unix HOWTO, Background, Sources of Design and Implementation Guidelines”, http://www.linux-tutorial.info/modules.php?name=Howto&pagename=Secure-Programs-HOWTO/sources-of-guidelines.html
[Microsoft] “Security in Software Localization”, Mohamed Elgazzar, https://msdn.microsoft.com/en-us/goglobal/bb688162.aspx
[Michael] “Defend Your Code with Top Ten Security Tips Every Developer Must Know”, Howard, Michael and Brown, Keith, https://blogs.msdn.microsoft.com/laurasa/2012/07/25/defend-your-code-with-top-ten-security-tips-every-developer-must-know/
[Msdn] “Guidelines for Writing Secure Code”, https://msdn.microsoft.com/en-us/library/ms182020.aspx
[StaticAnalysis] “List of tools for static code analysis”, https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
[TOCTOU] “Time of check to time of use”, https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use
[Top25] “2011 CWE/SANS Top 25 Most Dangerous Software Errors”, The MITRE Corporation, http://cwe.mitre.org/top25/
[ThreatModeling] “Threat Modeling for Modern System Firmware”, Robert P Hale, Vincent Zimmer, July 2013, http://www.uefi.org/sites/default/files/resources/Intel-UEFI-ThreatModel.pdf
[UHH] “Unix Hater’s Handbook” Simson Garfinkel, Daniel Weise, and Steven Strassman, http://web.mit.edu/simsong/www/ugh.pdf
[Wheeler] “Secure Programming for Linux and Unix HOWTO -- Creating Secure Software”, David Wheeler, http://www.dwheeler.com/secure-programs/

Authors

Jiewen Yao (jiewen.yao@intel.com) is EDKII BIOS architect, EDKII FSP package maintainer, EDKII TPM2 module maintainer, EDKII ACPI S3 module maintainer, with Software and Services Group at Intel Corporation. Jiewen is member of UEFI Security Sub-team and PI Security Sub-team in the UEFI Forum.

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