Finding SMM Privilege-Escalation Vulnerabilities in UEFI Firmware with Protocol-Centric Static Analysis

Jiawei Yin*‖††, Menghao Li*§‖††, Wei Wu‡, Dandan Sun*‖††, Jianhua Zhou*‖††, Wei Huo*§‖††, Jingling Xue†,
Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*, Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences‖, Beijing Key Laboratory of Network Security and Protection Technology††, UNSW Sydney†, Huawei Technologies‡
limenghao, huowei@iie.ac.cn

***Abstract*—The Unified Extensible Firmware Interface (UEFI) provides a specification of the software interface between an OS and its underlying platform firmware. The runtime services provided are seemingly secure as they reside in System Management Mode (SMM) at ring -2, assuming a higher privilege than the OS kernel at ring 0. However, their software vulnerabilities are known to be exploitable to launch ring 0 to ring -2 privilege escalation, i.e., SMM privilege escalation attacks.**

In this paper, we introduce an effective static analysis framework for detecting SMM privilege escalation vulnerabilities in UEFI firmware. We present a systematic study of such vulnerabilities and identify their root causes as being two types of references that can escape from the SMRAM, legacy references and unintentional references. Existing static analyses are ineffective in detecting such vulnerabilities in stripped COTS UEFI firmware images, which are developed based on a customized callback mechanism that organizes callable functions into protocols identified by GUIDs. By leveraging such a callback-based programming paradigm, we introduce SPENDER, the first static detection framework, which is founded on a novel protocol-centric analysis, for uncovering the potential SMM privilege escalation vulnerabilities in UEFI firmware efficiently and precisely. For a total of 1148 UEFI binaries collected from eight vendors, SPENDER has successfully found 36 SMM privilege escalation vulnerabilities (two 1-day and 34 0-day vulnerabilities), which can cause arbitrary code execution and arbitrary address write (and can thus enable, e.g., the attackers to install a bootkit into a flash drive). We have reported these 36 vulnerabilities to the vendors, with the two 1-day vulnerabilities confirmed as known previously but the 34 0-day vulnerabilities confirmed as new.

***Index Terms*—UEFI; SMM; Static Analysis; Protocol-Centric Analysis; Privilege Escalation Vulnerabilities;**

I. Introduction

Security Risks in UEFI Firmware at Runtime. The Unified Extensible Firmware Interface (*UEFI*) is currently the most popular industry standard that defines the software interface between an operating system and its underlying platform firmware. UEFI, which is a replacement of the older BIOS firmware, was shipped in more than 2.3 billion personal computer devices around the world from 2013 to 2021 [1].

To offer low-level control to peripheral hardware, UEFI firmware provides a number of runtime services for the operating system, such as managing system power, writing SPI flash and handling hardware errors. These runtime functionalities are provided by System Management Interrupt (*SMI*) handlers on Intel- and AMD-based platforms. However, the ability to control security-critical hardware this way makes it imperative to assess the security risks in SMI handlers.

Unfortunately, recent studies have demonstrated that software vulnerabilities in SMI handlers have been actively exploited to attack real-world users. By exploiting the vulnerabilities in SMI handlers, e.g., Aptiocalypsis [2] and ThinkPwn [3], the adversaries who attain the root privilege of a computer system, are able to modify the code and data in UEFI firmware, which is forbidden even for the root user, or install a UEFI bootkit [4] permanently, which is done to achieve the ultimate goal of compromising a system.

SMM Privilege Escalation Vulnerabilities. To secure the runtime services provided by UEFI firmware, a hardware isolation mechanism, together with an access control, i.e., a memory region (*SMRAM*) dedicated to SMI handlers and a privileged CPU execution mode (known as *system management mode (SMM)*) used for accessing this memory region, are introduced in modern computer architectures. However, this security solution still exposes a severe attack surface, i.e., *privilege escalation from kernel mode to SMM*. However, the SMM privilege escalation vulnerabilities that exist in UEFI firmware images have not been adequately studied so far.

Currently, software-based mitigation mechanisms such as DEP/NX [5] and Code Access Check (much like Supervisor Mode Execution Prevention (SMEP)) [6], [7] are bypassable [7], [8] by exploiting, for example, jump-oriented programming [9]. As for hardware-assisted mitigation mechanisms such as Intel's STM [10], we have not found even a single one (among the large number of UEFI firmware images analyzed) that supports the STM or other STM-like monitors. To detect SMM privilege escalation vulnerabilities, we are aware of only one previous work [11], which relies on a dynamic analysis by combining symbolic execution, fuzzing and concrete testing, making it ineffective due to the need for emulating a UEFI environment that is heavily hardware-dependent.

This Work. In this paper, we present a systematic study on SMM privilege escalation vulnerabilities that exist in UEFI firmware. We find that such vulnerabilities arise due to the

§Corresponding Author

data sharing between the operating system and an SMI handler invoked. From the perspective of the SMI handler that runs at a higher privilege (at ring -2), a shared data that can be controlled at ring 0 is deemed untrusted. If the data, without being carefully sanitized, eventually influences the control flow or data flow of the SMI handler, then it may be used to hijack the SMI handler, resulting in privilege escalation.

In addition, we have also developed a deep understanding about the execution model of an SMI driver/handler based on the UEFI specification [12]. Despite of hardware isolation and mitigation mechanisms deployed in modern CPUs, we find that data sharing is quite prevalent, with the data initially originating at a non-SMRAM memory region but subsequently referenced by the data inside SMRAM. Unfortunately, such references, called *escaped references* in this paper, are permitted by the UEFI specification [12]. No security risks can happen if all such escaped references are well-behaved, set up intentionally to do the right thing. In practice, however, an escaped reference may introduce high-risk data accesses for an SMI handler, violating its security requirement. In this paper, we have identified two types of escaped references, *legacy references* and *unintentional references*, depending on how they occur during the execution of an SMI handler.

Furthermore, we will give a formal definition of SMM privilege escalation vulnerabilities and use it to drive the development of a static analysis framework for detecting such vulnerabilities, which are caused by two types of escaped references. As UEFI firmware images are stripped binaries, which are developed based on a customized callback-based programming paradigm that organizes callable functions into protocols identified by GUIDs (Globally Unique IDentifiers), existing general-purpose static analyses [13], [14] for stripped binaries are ineffective due to their inability in discovering protocol callbacks accurately (Section II-B).

By exploiting this customized protocol-based callback mechanism in a novel way, we introduce SPENDER, a static protocol-centric analysis developed on top of *IDA Pro* [15] and *Angr* [16], [17], [18], to uncover potential SMM privilege escalation vulnerabilities. To the best of our knowledge, SPENDER represents the first static approach for detecting such vulnerabilities hidden in real-world UEFI firmware images efficiently and precisely. The basic idea behind our protocol-centric analysis can also be applied to other callback-rich programs, e.g., windows RPC [19] and COM [20].

Finally, we have applied SPENDER to 28449 SMI handlers extracted from 1148 UEFI firmware images (latest releases) from eight OEMs, covering all the three major IBVs (Insyde, AMI, and Phoenix). SPENDER is effective as it detects 36 vulnerabilities (including two 1-day vulnerabilities and 34 0-day vulnerabilities). SPENDER is precise as it exhibits a low rate of false positives at 10% only. SPENDER is efficient as it analyzes an SMI handler in 12.46 seconds, on average.

In summary, we make the following contributions:

- We present a systematical study of SMM privilege escalation vulnerabilities (from OS at ring 0) in UEFI firmware and identify two types of escaped references, legacy references and unintentional references, as their root causes.
- We introduce the first effective static detection framework, which is founded on a novel protocol-centric analysis, for detecting SMM privilege escalation vulnerabilities efficiently and precisely, by leveraging the customized callback mechanism adopted in developing UEFI firmware.
- We report an extensive evaluation of SPENDER in detecting 36 vulnerabilities (including two 1-day and 34 0-day vulnerabilities) in 1148 UEFI firmware images. All these 36 vulnerabilities have been confirmed as privilege escalation vulnerabilities by the vendors, with the two 1-day vulnerabilities as being known previously, but the 34 0-day vulnerabilities as new (of which 28 have been repaired with CVE IDs credited and the remaining 6 are being repaired).

II. Background

A UEFI firmware image [21] contains five major modules: the Security (SEC) module, the Pre-EFI Initialization (PEI) module, the Driver Execution Environment (DXE) module, the SMM module, and the Boot Device Selection (BDS) module. In this paper, we focus on its SMM module only. In addition to boot a computer system securely, the UEFI firmware provides a number of runtime services via its SMM module for the operating system to control flexibly the low-level peripheral hardware that is also security-critical to the entire system.

To keep a computer system secure in the presence of a compromised operating system, a hardware isolation mechanism, together with an access control, is designed for using the runtime services in Intel- and AMD-based platforms. To facilitate this isolation, system management mode (*SMM*), a higher privileged CPU execution mode [22] than kernel mode, is provided. Meanwhile, a dedicated memory region, called system management RAM (*SMRAM*), is reserved from the memory space, which cannot be accessed or modified from the kernel. The SMRAM is accessed by virtual addresses via a separate, dedicated page table residing in the SMRAM [23]. The programs running in SMM are required to reside within the SMRAM, but capable of accessing the entire physical memory according to the UEFI specification [12].

A. SMM Drivers

An SMM module consists of a number of SMM drivers. Each SMM driver consists of usually three components: (1) a set of protocols, (2) a group of SMI handlers, and (3) an initialization function. A *protocol* is a vendor-defined interface. In general, its function pointer fields point to some exported functions (*protocol functions*) of its containing SMM driver and its data fields are used to communicate its data with other SMM drivers. Each protocol is associated with an identifier, encoded as a 128-bit Globally Unique IDentifier (GUID) string. An *System Management Interrupt (SMI) Handler* represents a runtime service implemented by calling some exported functions in some protocols (from possibly many SMM drivers). Finally, an SMM driver uses its initialization function to register and/or retrieve all the protocols concerned.

Typically, a group of runtime services providing similar functionalities (e.g. SPI flash management, power management, or graphic device management) are packaged into the

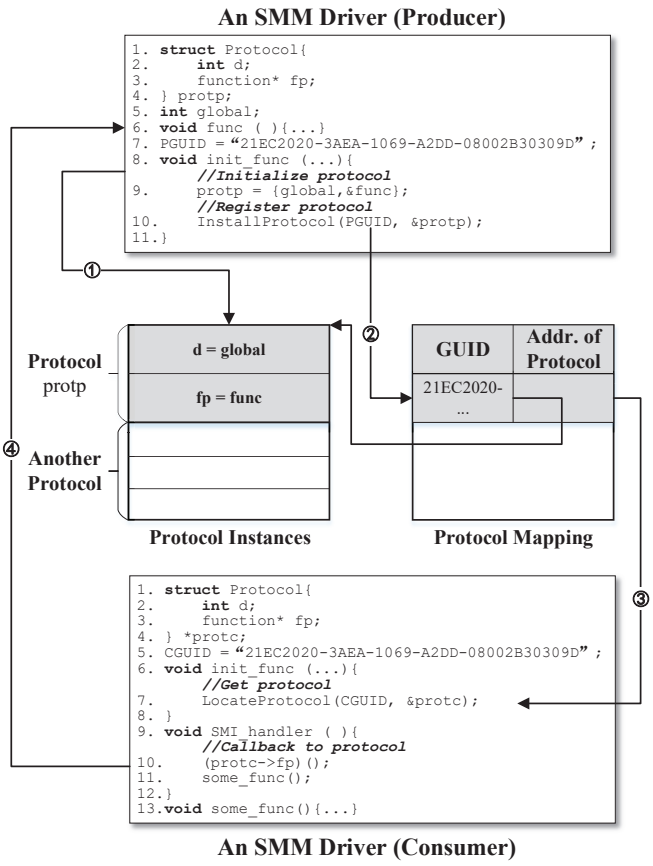

Figure 1: Programming paradigm of SMM drivers.

same SMM driver, which is implemented in C and then compiled into one EFI file in portable executable format.

1) Programming Paradigm: Figure 1 illustrates how two SMM drivers (which are not necessarily distinct) interact via protocols identified by their GUIDs in a callback-based programming paradigm. The top SMM driver calls its own `init()` function to advertize a protocol identified by a GUID as "21EC2020-3AEA-1069-A2DD-08002B30309D" (①) and register this protocol in a so-called protocol-mapping table (②). The bottom SMM driver calls its own `init()` function to locate this same protocol provided by the top SMM driver from the protocol mapping table (③), so that its own SMI handler, named as `SMI_Handler()`, can provide its runtime service by using a so-called *protocol callback* (④)

Protocols enable inter-module communication in UEFI across different modules (SMM drivers or some other client applications). SMM drivers are both producers and consumers of protocols. In this simple illustrating example, the top SMM driver produces the protocol "`protp`" identified by a GUID as "21EC2020-3AEA-1069-A2DD-08002B30309D" and the bottom SMM driver consumes this protocol.

2) Execution Model: Figure 2 illustrates the execution model of an SMM driver, which is divided into two phases, the *boot phase* and the *runtime phase*.

The Boot Phase. During this phase, as shown in Figure 2(a), the UEFI firmware will perform a series of three initialization operations by using its DXE module (via its initialization function) at ring 0. First, the DXE module allocates and initializes data objects (referred to as *boot-only objects*) at a certain memory area, denoted DXE-DATA (①), which resides outside the SMRAM. Second, the DXE module makes use of these data objects to call an initialization function in the SMM module, which loads all the SMM drivers into the SMRAM and executes their initialization functions (②). Finally, the DXE module locks down the SMRAM at ring -2 to prevent it from being accessed by the OS kernel from now on (③).

We write SMM-DATA and SMM-CODE to denote the data and code sections in the SMRAM, respectively.

The Runtime Phase. During this phase, as shown in Figure 2(b), an SMI handler will interact with the operating system in a series of three steps just before it is invoked.

In Step 1 (①), the kernel prepares the execution context for the SMI handler by storing the context information (e.g., the ID of the destination SMI handler, which is "12th" for illustration purposes) into *CommBuffer*, a data object allocated in DXE-DATA and designated to pass data to the SMI handler. The kernel then sends an interrupt instruction with the given SMI handler ID. In Step 2 (②), the kernel sets up this particular execution context for the SMI handler to be invoked. Upon receiving the interrupt instruction, the CPU will switch to SMM (at ring -2 on Intel-like platforms) and then invoke a special *SMM dispatcher* function, which will allocate and initialize two data objects in the SMRAM. One object, named *CommBuffer(S)*, stores a copy of *CommBuffer* to be consumed by the SMI handler. The other, named *SavedState*, stores the CPU context (e.g., for general-purpose registers), which will be recovered when the SMI handler returns. In Step 3 (③), the SMM dispatcher calls the SMI handler as desired.

B. Detecting SMM Privilege Escalation Vulnerabilities

In the past few years, some SMM privilege escalation vulnerabilities have been reported in a few media releases [6], [7], [8]. Such vulnerabilities remain exploitable even though some software-based mitigation mechanisms such as DEP/NX (by making data non-executable) [5] and Code Access Check (much like Supervisor Mode Execution Prevention (SMEP)) [6], [7], [8] are deployed, by exploiting jump-oriented programming [9]. Intel has proposed several hardware-assisted mitigation mechanisms (e.g., STM [10]). However, as mentioned in its 2020 white paper [5], "most current BIOS's do not support the STM or other STM monitors." For the large number of UEFI firmware images (totaling 1148) that we have analyzed in this research, we have not found even a single one that supports the STM or other STM-like monitors.

Therefore, the problem of developing program analysis techniques for detecting SMM privilege escalation vulnerabilities becomes imperative. There is only one prior attempt [11], which performs its vulnerability detection dynamically by combining symbolic execution, fuzzing and concrete testing, making it ineffective due to the need for emulating a UEFI environment that is heavily hardware-dependent.

In this research, we aim to develop an effective static analysis framework that can perform its vulnerability detection in stripped COTS UEFI firmware images efficiently and precisely. The main challenge lies in how to identify protocol

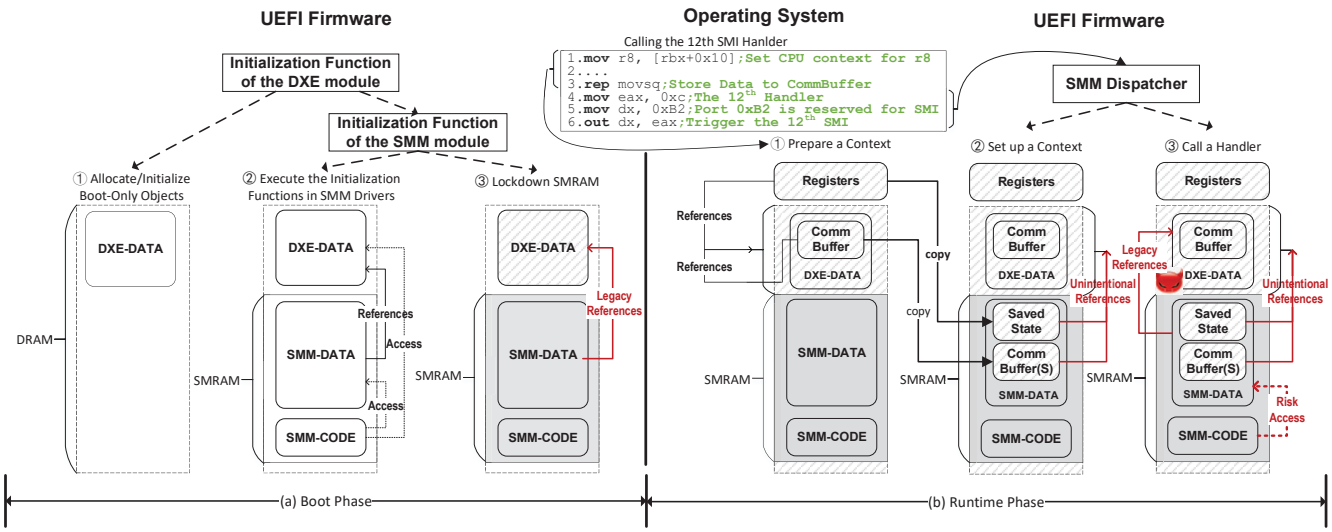

Figure 2: Execution model of SMM Drivers (with two phases).

callbacks, i.e., protocol functions called indirectly at protocol-related indirect call instructions in the SMM drivers (e.g., protc−>fp() at Line 10 in Figure 1) efficiently and precisely. For the SMM drivers implemented in a callback-based programming paradigm, their protocols (identified by GUIDs) are usually installed into and retrieved from a protocol mapping table (i.e., an array) (involving a series of if-then-else statements, which operate on void pointers, in a path-sensitive manner). Existing static analyses for stripped binaries are ineffective in identifying protocol callbacks in UEFI firmware, as they will either miss excessively many real targets unsoundly as in *IDA Pro* [15] and *Angr* [16], [17], [18] or introduce excessively many false targets imprecisely as in [13], [14] (due to the lack of path-sensitivity) and in [24], [25], [26] (due to the need for enforcing control-flow integrity correctly).

.

III. Understanding SMM Privilege Escalation Vulnerabilities and Their Root Causes

An SMM privilege escalation vulnerability arises when a (compromised) kernel-space program accesses/modifies the resources in SMM (system management mode) arbitrarily or executes certain code that should have been launched in SMM unintendedly. By exploiting such a vulnerability, a malicious kernel-space program can inject a shellcode into the SMRAM or reuse code sequences, i.e., JOP gadgets, in the SMRAM to reflash the UEFI firmware or subvert the TPM [27].

We conduct a systematic study on SMM privilege escalation vulnerabilities. In Section III-A, we discuss the security principles behind the UEFI specification [12]. In Section III-B, we identify two types of escaped references (as highlighted in Figure 2) as the root causes of SMM privilege escalation vulnerabilities based on the execution model of an SMM driver (reviewed in Section II-A2). In Section III-C, we describe our insights on formulating the problem of detecting such vulnerabilities (highlighted as "Risky Access" in Figure 2) as an escaped-references-driven value-flow analysis. Finally, in Section III-D, we discuss our threat model used.

A. Security Principles for Developing SMM Drivers

As UEFI firmware can access the entire physical main-memory space (Section II-A)), the behavior of an SMI handler must be constrained rigorously. Therefore, the UEFI specification defines the following two design principles:

- [PRINCIPLE-I] [12]: an SMI handler should not be allowed to access a memory area outside the SMRAM.
- [PRINCIPLE-II] [5]: an SMI handler is responsible for sanitizing carefully the untrusted data coming from the operating system or the virtual machine manager.

[PRINCIPLE-I] serves to constrain the range of memory addresses accessed by an SMI handler. However, this alone is not sufficient for protection. As some untrusted data (e.g., those in *CommBuffer(S)*) can be copied from a lower privileged memory area into the SMRAM (Section II-A2), the SMI handler must also sanitize such untrusted data before referencing it according to [PRINCIPLE-II]. Otherwise, a code reuse attack can happen via jump-oriented programming [6] (even in the presence of DEP/NX [5] and Code Access Check [6], [7], [8]) when the SMI handler, e.g., makes an indirect call on a non-trusted function pointer that points to the address of its first JOP gadget) even though its address falls within the SMRAM.

While [PRINCIPLE-II] is straightforward for UEFI firmware developers to follow, the situation for [PRINCIPLE-I] becomes significantly challenging, since an SMI handler has the ability to access the entire physical memory space. After having examined the execution model of an SMM driver (Figure 2), we find that escaped references (i.e., the references made from inside the SMRAM to outside) are the culprits for violating [PRINCIPLE-I]. If an attacker who has gained the root privilege of a computer system controls such escaped references, an SMM privilege escalation attack can occur.

B. Escaped References from the SMRAM

For an SMM driver, its data and code are loaded into SMM-DATA and SMM-CODE in the SMRAM, respectively (Figure 2). According to [PRINCIPLE-I], an SMI handler

The DXE Module

```
1. struct EFI_SYSTEM_TABLE {
2.     ...
3.     EFI_BOOT_SERVICES BootServices;
4.     EFI_RUNTIME_SERVICES RuntimeServices;
5. } SystemTable;

6. void init_func(...){
7.     ...
8.     /* Call init_func of an SMM Driver,
          Use SystemTable as the 2nd argument */
9. }
```

An SMI Handler

```
1. struct EFI_BOOT_SERVICES *BootServices;

2. void init_func(EFI_HANDLE IH, EFI_SYSTEM_TABLE * ST){
3.     BootServices = ST->BootServices;
4.     BootServices-> InstallProtocol(...);
5. }
6. void SMI_handler(){
7.     BootServices->LocateProtocol(...);
8. }
```

Figure 3: A legacy reference BootServices.

should access SMM-DATA and SMM-CODE only. However, we find that this principle can be violated by two types of escaped references, legacy and unintentional references, which are created during the boot and runtime phases of an SMM driver, respectively, as explained below.

1) Legacy References: Legal references are created during the boot phase of an SMM driver (Figure 2(a)). A *legacy reference* is created if an object in the SMRAM remains to point to an object in DXE-DATA, which resides outside the SMRAM, at the end of the boot phase of an SMM driver. According to the UEFI specification [12], the SMRAM must be locked down at ring -2 at the end of this boot phase, but how to deal with such legacy references afterwards is unspecified. If an SMI handler accesses an object in DXE-DATA via a legacy reference, [PRINCIPLE-I] will be violated, as DXE-DATA is controllable at ring 0 by the kernel.

We argue that legacy references should be avoided. However, such a practice is neglected, as shown by a real SMM handler (from Lenovo's UEFI firmware for ThinkPad T450) in Figure 3. In this handler, BootServices (a legacy reference) is initialized as a field of SystemTable, which resides in DXE-DATA, at Line 3, and then dereferenced at line 7, resulting in an SMM privilege escalation vulnerability.

2) Unintentional References: Unintentional references are created during the runtime phase of an SMM driver (Figure 2(b)). When setting up the execution context for the SMI handler to be invoked, the UEFI specification [12] does not dictate how the SMM dispatcher should copy *CommBuffer* in DXE-DATA to *CommBuff(S)* in the SMRAM. Usually, a shallow copy is made. Thus, a pointer in *CommBuffer* will be copied to *CommBuff(S)*, causing the pointer in *CommBuff(S)* to point to an object outside the SMRAM. Such a reference made from an object in *CommBuff(S)* to an object outside the SMRAM is called an *unintentional reference*. Similarly, unintentional references can also be created when general-purpose registers are copied into *SavedState* in the SMRAM.

If an SMI handler dereferences an unintentional reference in *CommBuff(S)* or *SavedState*, [PRINCIPLE-I] will be violated, as the dereferenced object is controllable at ring 0 by the kernel. Consider a real SMM handler (from Dell's UEFI

A Code Snippet in the Kernel Space

```
1. struct CommDataType {
2.     int field;
3.     int status;
4. } commData;

5. void func( ){
6.     __asm{lea rbx, commData};
7.     ...
       //Trigger an SMI
8.     __asm{out dx, eax};
9. }
```

An SMI Handler

```
1. struct CommDataType {
2.     int field;
3.     int status;
4. };

5. void SMI_handler(){
6.    struct CommDataType * data;
      // Get the value of rbx from SavedState to data
7.    ReadSaveState(...,rbx_index, data);
8.    ...
9.    data->status = 0;
10. }
```

Figure 4: An unintentional reference data.

firmware image for G7-17) in Figure 4. In the SMI handler, data at Line 7 receives the value of rbx, which is the address of commData at Line 6 in the kernel code stored in *SavedState*, resulting in an SMM privilege escalation vulnerability.

Finally, we argue that the semantic gap between the operating system and the *SMM dispatcher* makes it impractical to deep-copy *CommBuffer* into *CommBuff(S)*, as both run in two different privilege modes. In general, it can be difficult for the *SMM dispatcher* to distinguish pointer data from non-pointer data in *CommBuffer* (or registers). This suggests that pointers should never be stored in *CommBuffer* or registers in order to avoid introducing unintentional references accidentally. However, such a practice is also not followed, in practice.

C. Key Insights for Vulnerability Detection

We can formulate the problem of finding SMM privilege escalation vulnerabilities as a value-flow analysis that tracks the flow of the objects pointed to by escaped references.

An instruction is *security-sensitive* if it operates on some potentially kernel-controllable operand. There are two types of security-sensitive instructions: (1) *indirect memory write instructions*, such as $mov\ [rax],\ rbx$), and (2) *indirect control-transfer instructions*, such as $jmp\ [rax]$ and $call\ [rax]$, where the target address operand in $[rax]$ in each of these instructions is considered to be *potentially kernel-controllable*.

An SMM privilege escalation vulnerability may occur *only if* two conditions are met: (1) a security-sensitive instruction operates on a potentially kernel-controllable operand that is pointed to by an escaped reference (by violating [PRINCIPLE-I]), and (2) the value-flow path for propagating the operand to the security-sensitive instruction lacks a sanity check and is thus deemed unsafe (by violating [PRINCIPLE-II] too).

D. Threat Model

We assume an attacker with the root privilege of a computer system. The attack goal is to bypass the security boundaries set by the UEFI firmware to leak/modify sensitive data in SPI

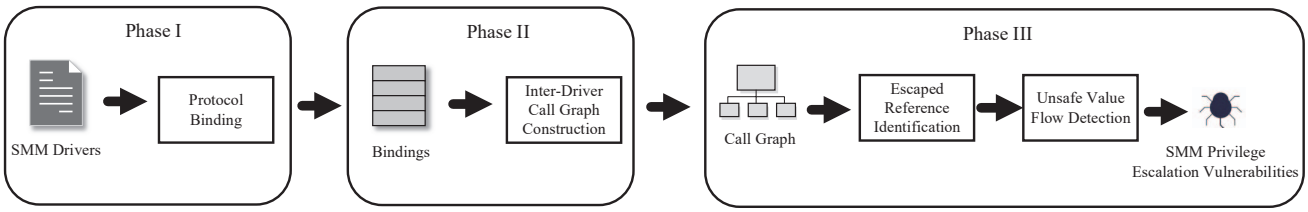

Figure 5: The high-level work-flow of SPENDER.

flash (e.g., BIOS password) or to install a bootkit that will be alive even after a reinstallation of the operating system.

We assume further that software-based mitigation techniques, such as DEP/NX [5] and Code Access Check [6], [7], [8], are fully deployed. As discussed earlier in Section II, such mitigation techniques are bypassable as the attacker (with the root privilege of a computer system) can still launch SMM privilege escalation attacks by exploiting jump-oriented programming [9] (without having to rely on code injection).

IV. SPENDER: Protocol-Centric Static Analysis

We introduce SPENDER, an effective static protocol-centric analysis for detecting SMM privilege escalation vulnerabilities in the set of SMM drivers appearing in a stripped COTS UEFI firmware image effectively and precisely. As depicted in Figure 5, SPENDER proceeds in three phases. In *Phase I: "Protocol Binding"*, SPENDER relates a protocol pointer in a consumer SMM driver to its corresponding protocol (callback) installed in a producer SMM driver, and at the same time, produces also an *inter-driver call graph* for the set of SMM drivers analyzed without any inter-driver call graph edge yet. In *Phase II: "Inter-Driver Call Graph Construction"*, SPENDER augments this call graph by adding all the missing inter-driver call graph edges, thereby obtaining a complete inter-driver call graph. In *Phase III: "Vulnerability Detection"*, SPENDER performs an inter-procedural static analysis over the final call graph thus constructed to detect the SMM privilege escalation vulnerabilities in the set of SMM drivers given. SPENDER works context-sensitively (by distinguishing calling contexts) and flow-sensitively (by distinguishing the flow of control), tracking of the flow of the objects pointed to by escaped references to all the security-sensitive instructions (by verifying [PRINCIPLE-I]), filtering out the paths with sanity checks (by verifying [PRINCIPLE-II]), and reporting the remaining potential vulnerabilities.

The key novelty behind our protocol-centric analysis for a UEFI firmware image is to decouple the problem of discovering its inter-driver call graph edges from the problem of detecting its vulnerabilities. By relating a consumer protocol to its producer protocol via the same GUID used in Phase I, we are able to discover the protocol functions called at protocol-related indirect call instructions precisely in a path-sensitive manner in Phase II. Given the precise call graph built this way, SPENDER can then conduct a context- and flow-sensitive analysis for detecting vulnerabilities by solving a taint analysis (with escaped references as the taint sources) in Phase III.

SPENDER is expected to have many other applications. By discovering protocol callbacks precisely, SPENDER can build a call graph precisely across the SMM drivers, enabling other whole-program data-flow analyses to be performed, including inter-procedural alias analysis and buffer overflow detection.

A. Protocol Binding

In UEFI, protocols enable inter-driver communication with each protocol being produced by one SMM driver and consumed by several other SMM drivers. In this phase, SPENDER is responsible for relating a protocol pointer used in a LocateProtocol() call in a consumer to the corresponding protocol installed by an InstallProtocol() call in a producer, by leveraging the fact that both calls are identified by the same GUID. In addition, this phase will also produce an incomplete call graph for the set of SMM drivers analyzed in the sense that all inter-driver call graph edges are still absent.

Consider Figure 1, where the top producer SMM driver allocates and initializes a protocol by invoking an API function, InstallProtocol(), with its arguments being a GUID and a protocol (protp), and the bottom consumer SMM driver initializes a pointer (protc) that points to a protocol by invoking an API function, LocateProtocol(), with its arguments being a GUID and a protocol pointer (protc). SPENDER will bind protc to protp by connecting a pair of such API function invocations that use the same GUID, which is discovered from the data section SMM-DATA in the SMRAM.

As shown in Algorithm 1, SPENDER takes as input a set of SMM drivers and produces as output an initially incomplete (inter-driver) call graph and a set of protocol bindings for these SMM drivers. The **for** loop at Lines 5-16 processes all the SMM drivers in turn. For each SMM driver *sd*, SPENDER first lifts its binary representation to a so-called VEX IR by applying *Angr* [16] (Lines 7-8). In addition, SPENDER also recovers the call stack layout for each call site in *sd* by following [28] in order to obtain the prototype (i.e., signature) of any function invoked. SPENDER then builds an initially incomplete call graph for *sd* by resolving indirect calls (except for all the protocol-related inter-driver calls) using *Angr* (Lines 11-12). Finally, SPENDER discovers all the GUIDs in *sd* from the data section SMM-DATA in the SMRAM according to their syntactic format specified by IETF in RFC 4122 [29], expressed as a set of $[id, addr]$ pairs, where *id* is a GUID string and *addr* is its address stored in SMM-DATA (Lines 15-16).

Next, SPENDER identifies the producer and consumer protocols in all the SMM drivers (Lines 19-30). To find all the call sites that invoke InstallProtocol() and LocateProtocol() without knowing their symbols in stripped binaries, SPENDER leverages the fact that each of these two API functions

Algorithm 1 Protocol Binding

```
1: In: SMMdrivers (set of SMM drivers)
2: Out: (1) CG (the call graph for the SMM drivers)
3:       (2) ProtBindings (set of protocol bindings)
4: procedure BIND_PROTOCOL(SMMdrivers)
5:     for sd ∈ SMMdrivers do
6:         // 1. Lift to the VEX IR and recover the call stack for sd
7:         ir ← Lift(sd)
8:         VEXIR ← VEXIR ∪ {ir}
9:
10:        // 2. Build the initial call graph for sd
11:        cg ← Build_CG(ir)
12:        CG ← CG ∪ {cg}
13:
14:        // 3. Find the GUID strings and their addresses in sd
15:        GUIDsAndAddrs ← Locate_GUIDs(sd)
16:        SetofGUIDs ← SetofGUIDs ∪ GUIDsAndAddrs
17:
18:    // 4. Identify producer and consumer protocols
19:    for f ∈ CG do
20:        for inst ∈ f do
21:            if inst has the form "mov reg, addr" then
22:                if [id, addr] ∈ SetofGUIDs then
23:                    call ← Locate_Callsite(inst)
24:                    sig ← Recognize_Function(call)
25:                    if sig = "InstallProtocol" then
26:                        addrp ← address of its protocol installed
27:                        ProtObjs[id] ← addrp
28:                    else if sig = "LocateProtocol" then
29:                        addrc ← address of its protocol pointer
30:                        ProtPtrs[id] ← ProtPtrs[id] ∪ {addrc}
31:
32:    // 5. Bind protocol pointers to their protocols
33:    for id such that ProtObjs[id] ≠ ∅ do
34:        for addrc ∈ ProtPtrs[id] do
35:            ProtBindings ← ProtBindings ∪ {[addrc, ProtObjs[id]]}
```

takes a GUID as an argument. Therefore, SPENDER traverses all the functions in the SMM drivers (Lines 19-20) and finds all the instructions that use the GUID addresses stored in *SetofGUIDs*. If such an instruction, denoted *inst*, is found (Line 21), which uses $[id, addr]$ (Line 22), the function Locate_Callsite() at Line 23 is called to find the call instruction (which is unique due to the x86-64 calling convention used) nearby that uses *inst* as its argument. Given the call stack layout recovered earlier for the call site, the function Recognize_Function() is called at Line 24 to infer the prototype of any function invoked, denoted *sig*. We then record in $ProtObjs[id]$ the address of the protocol installed in a producer if *sig* is InstallProtocol() (Lines 25-27) and in $ProtPtrs[id]$ the address of the protocol pointer used in a consumer if *sig* is LocateProtocol() (Lines 28-30).

Finally, SPENDER obtains *ProtBindings*, which represents the bindings of the protocol pointers in consumers with their corresponding protocols in producers (Lines 33-35). At the end of the first phase, the call graph for the set of SMM drivers is incomplete. All the inter-driver call graph edges are missing but will be added by SPENDER in its second phase.

Figure 6 illustrates how SPENDER performs its protocol binding during its first phase for the stripped binary of the simple program in Figure 1. For readability, Figure 6(a) gives its two SMM drivers in the form of x86-64 assembly instructions. Figure 6(b) depicts the inter-driver call graph constructed for both SMM drivers, which consists of the two disconnected call graphs for the two SMM drivers, where the inter-driver call graph edge highlighted in red is still missing but will be discovered by SPENDER during its second phase. After having located PGUID = CGUID = "21EC2020-3AEA-1069-A2DD-08002B30309D", SPENDER finds its two users, the instruction at Line 7 in the producer and the instruction at Line 3 in the consumer. SPENDER will then find their corresponding call cites, with one at Line 9 in the producer for calling InstallProtocol() and the other at Line 5 in the consumer for calling LocateProtocol(). Finally, SPENDER will produce one protocol binding, by binding protc to protp.

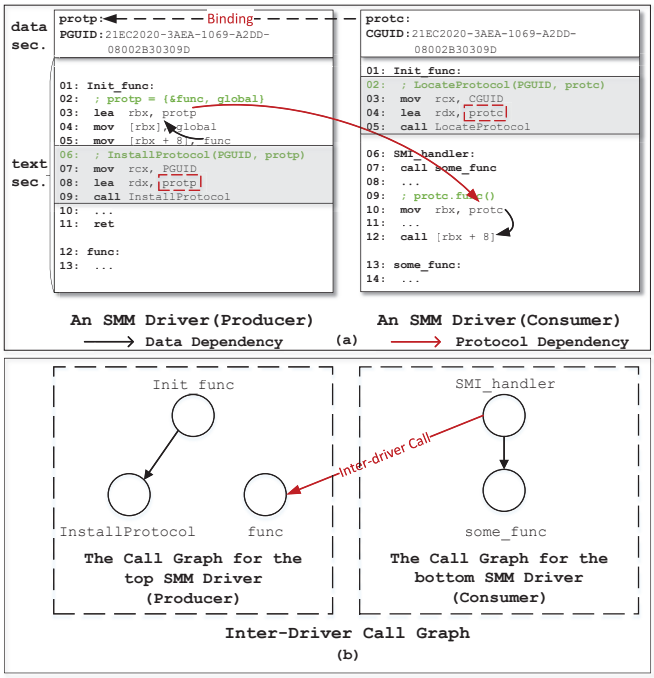

Figure 6: The protocol-centric analysis applied to the program given in Figure 1. For readability, the stripped binaries for its two SMM drivers are given in assembly instructions.

During its second phase, SPENDER will resolve the so-called protocol dependency (highlighted in red in Figure 6(a)), and consequently, be able to add the missing inter-driver call graph edge (also highlighted in red in Figure 6(b)).

B. Inter-Driver Call Graph Construction

In this second phase, SPENDER will complete the inter-driver call graph built in the first phase by adding the missing inter-driver call graph edges for all the protocol-related indirect calls by considering not only the traditional data dependencies but also the so-called protocol dependencies. An instruction *instc* that references an address *addrc* is said to have a *protocol dependency* on another instruction *instp* that references an address *addrp* if $[addrc, addrp] \in ProtBindings$.

As shown in Algorithm 2, SPENDER takes as input *CG* (an incomplete inter-driver call graph) and *VEXIR* (an intermediate representation) for a given set of SMM drivers constructed during the first phase and produces as output the final call

Algorithm 2 Inter-Driver Call Graph Construction

```
1: In: (1) CG (the call graphs for the SMM drivers)
2:     (2) VEXIR (the IR for the SMM drivers)
3: Out: CG (an inter-driver call graph for the SMM drivers)
4: procedure CONSTRUCT_IDCG(CG)
5:     // 1. Construct protocol dependencies
6:     for instc, instp ∈ VEXIR such that
           instc references an address addrc and
           instp references an address addrp do
7:         if [addrc, addrp] ∈ ProtBindings then
8:             Add a protocol dependency from instp to instc
9:
10:    // 2. Resolve protocol-related indirect calls
11:    for f ∈ CG do
12:        for inst ∈ f do
13:            if inst is of the form "call [reg+offset]" then
14:                if inst is not resolved then
15:                    sliceb ← Backward_Slice(inst, reg)
16:                    cend ← Last_Inst(sliceb)
17:                    if cend references no protocol pointers then
18:                        continue
19:                    pstart ← Protocol_Dependency(cend)
20:                    Let pstart be of the form "mov regp, addr"
21:                    pend ← Resolve(pstart, regp, offset)
22:
23:                    // 3. Add an inter-driver call graph edge
24:                    addr ← the target address of pend
25:                    Add a call graph edge from f to addr
```

graph that is still represented by *CG* except that all the inter-driver call graph edges have now been added.

SPENDER starts by discovering all protocol dependencies based on *ProtBindings* (Lines 6-8). Such protocol dependencies are critically important to enable us to resolve indirect call targets for all protocol-related calls successfully.

SPENDER then discovers the targets at all protocol-related indirect calls (Lines 11-25). For each unresolved indirect call instruction, denoted *inst*, which is of the form call [reg+offset] (Lines 11-14), SPENDER resolves its call target as follows. SPENDER conducts a backward slice from *inst* by using *reg* as the slicing criterion (Line 15). Let *cend* be the last instruction of the slice, where the definition of *reg* is found (Line 16). If *cend* does not reference any protocol pointer, we will continue to deal with the next unsolved indirect call instruction (Lines 17-18). Otherwise, let *pstart* be the instruction that has a protocol dependency with *pend* (Line 19), which must always have the form mov regp, addr given at Line 20 due to the way that a protocol is registered (i.e., initialized and installed). Then the function Resolve() is called to resolve the target for *inst*. In this function, we model a protocol as a data object with an unknown data structure as in [30]. We interpret reg+offset as representing a field of the protocol, where reg is a register and offset is a register or an integer constant. We then track the value flow forwards from *pstart* with this known offset until we see an instruction *pend* that references an address in the code section SMM-CODE in the SMRAM (Lines 21). If *addr* represents this target address (Line 24), then the target at *inst* has been found (Line 25).

Let us return to Figure 6, where the simple program in Figure 1 is analyzed. During this second phase, SPENDER will first discover the protocol dependency from Line 3 in the producer to Line 10 in the consumer, depicted by the red arrow in Figure 1(a). SPENDER will then work on discovering the target for the indirect call instruction at Line 12 in the consumer. Slicing from this instruction backwards will end at the instruction at Line 10 in the consumer. Following its protocol dependency, SPENDER is able to continue tracking the value flow forwards from the instruction at Line 3 in the producer, and finally, identifies the function func as the target at Line 12 in the consumer. As a result, an inter-driver call graph, as depicted by the red arrow in Figure 6(b), is found.

C. Vulnerability Detection

In this last phase, SPENDER will conduct an escape-references-driven value-flow analysis to detect SMM privilege vulnerabilities in UEFI firmware context- and flow-sensitively, as formulated earlier in Section III-C. As shown in Algorithm 3, SPENDER will first identify two types of escaped references and then carry out its value-flow analysis.

To identify legacy references (Lines 6-9), we only need to analyze the initialization functions of all the SMM drivers (Line 6), which are the entry points for some boot-only objects (residing in DXE-DATA in Figure 2) to "sneak into" some SMM drivers, ending up eventually to be still referenced there (directly or indirectly) even after the SMRAM has been locked down (as described in Section III-B1). For each SMM driver, its initialization function can be found easily since its address is recorded in the field AddressOfEntryPoint in the header part of its PE file. Given an initialization function f, SPENDER relies on Legacy_Refs(f) to identify its set of legacy references, denoted *LR* (Lines 7-9). Legacy_Refs(f) performs simply a standard context- and flow-insensitive inter-procedural mod-ref analysis on the call graph rooted by f (a sub-graph of *CG*). Then all the objects that reside in SMM-DATA (in the SMRAM) are deemed legacy references if they are used as pointers but not initialized (as they have been highly likely initialized to point to some boot-only objects by the DXE module during the boot phase of all SMM drivers (Figure 2)). In addition, an object that points to one of these objects directly or indirect is also deemed a legacy reference.

To identify unintentional references (Lines 11-22), we distinguish two cases, depending on whether they arise from *CommBuffer* or *SavedState* (Section III-B2). We only need to consider SMI handlers (identified easily at Line 11), the entry points for launching SMM privilege escalation attacks.

- ***CommBuffer.*** Locating an unintentional reference to *CommBuff(S)*, which is copied from *CommBuffer*, accessed in an SMI handler h is relatively straightforward (Lines 12-14). According to the function prototype of an SMI handler defined in the UEFI specification [12], the function Locate_Commbuffer(h) is simply called to identify *CommBuff(S)* as the third argument of h.
- ***SavedState.*** According to the UEFI specification [12], *SavedState* can only be accessed by its API function ReadSaveState(). By proceeding similarly as in the case of identifying InstallProtocol()/LocateProtocol() in its first phase (Algorithm 1), SPENDER identifies an unintentional reference to *SavedState* as follows (Lines 16-22).

Algorithm 3 Vulnerability Detection

```
1: In: CG (the call graph for the SMM drivers)
2: Out: Vuls (set of SMM privilege escalation vulnerabilities)
3: procedure DETECT_VUL(CG)
4:     // 1. Identify escaped references
5:     // (a) Legacy references
6:     InitFuns ← set of init functions of all SMM drivers
7:     for f ∈ InitFuns do
8:         LR ← Legacy_Refs(f)
9:         EspRefs ← EspRefs ∪ LR
10:    // (b) Unintentional references (CommBuffer)
11:    SMIHDLs ← set of SMI handlers (i.e., the functions
                 in CG but not in InitFuns with no callers)
12:    for h ∈ SMIHDLs do
13:        pBuffer ← Locate_Commbuffer(h)
14:        EspRefs ← EspRefs ∪ {pBuffer}
15:    // (c) Unintentional references (SavedState)
16:    for h ∈ SMIHDLs do
17:        for inst ∈ h do
18:            if inst is an indirect call (i.e.,"call [reg+offset]") then
19:                sig ← Recognize_Function(inst)
20:                if sig = "ReadSaveState" then
21:                    pState ← Locate_State(inst)
22:                    EspRefs ← EspRefs ∪ {pState}
23:
24:    // 2. Detect vulnerabilities
25:    Vuls ← EspRefs_Driven_Detection(CG, EspRefs)
```

For an indirect call instruction *inst* (Lines 17-18) in an SMI handler h, SPENDER analyzes the function prototype of its target (based on the call stack layout recovered for *inst*) to see if ReadSaveState() is actually called (Lines 19-20). If this is the case (Lines 21-22), SPENDER will identify the address of an object that records *SavedState* as an unintentional reference.

Finally, SPENDER will conduct a value-flow analysis to track the flow of the objects pointed by all escaped references recorded in *EspRefs* across the call graph represented by *CG* (Line 25), by solving it as essentially a taint analysis (with all the escaped references as the taint sources). We check whether [PRINCIPLE-I] is violated by restricting ourselves to escaped references. We check whether [PRINCIPLE-II] is violated by looking for the existence of sanity checks. To accomplish this, SPENDER leverages a heuristic code pattern introduced earlier [31], [32]. A code snippet with a branch instruction is treated as a sanity check if (1) one branch terminates program execution by returning an integer or string (regarded as an error code), and (2) the other branch continues program execution.

EspRefs_Driven_Detection(*CG*, *EspRefs*) employs a standard context- and flow-sensitive inter-procedural data flow analysis algorithm by performing also the standard summaries [33] to improve its efficiency without losing any precision. The analysis starts from the escaped references in *EspRefs*, tracks the value flow in *CG* forwards, and stops its value-flow propagation along a particular path if (1) a sanity check is encountered or (2) a security-sensitive instruction is encountered without any prior sanity check being seen. In Case (1), no warning is raised as [PRINCIPLE-II] is followed. In Case (2), a warning is raised as [PRINCIPLE-II] is not followed even though [PRINCIPLE-I] is violated.

V. Evaluation

We show that SPENDER represents an effective static analysis framework for detecting SMM privilege escalation vulnerabilities efficiently and precisely in stripped COTS UEFI firmware images. SPENDER is effective as it detects 36 vulnerabilities (including two previously known 1-day vulnerabilities and 34 new 0-day vulnerabilities) in stripped COTS UEFI firmware images. SPENDER is precise as it exhibits a low rate of false positives at 10%. SPENDER is efficient as it analyzes an SMI handler in 12.46 seconds, on average.

A. Experiment Setup

Implementation. We have built SPENDER on top of *IDA Pro* (an interactive disassembler) [15] and *Angr* (a binary analysis framework) [16], [17], [18], with about 5000 LOC in Python. We have done our experiments on a PC with a dual-core Intel Core i5-7260U CPU @2.20GHz with 8 GB of RAM running on Ubuntu 16.04. Its hard disk has been extended to 2TB in order to store the UEFI firmware images studied, together with the intermediate results and final reports generated.

Dataset Preparation. During July 2019 – August 2019, we crawled a total of 1148 UEFI firmware images from public official websites of eight popular OEMs including Lenovo, HP, Asus, Dell, Huawei, Intel, MSI and Microsoft. We manually examined these firmware images, ensuring they cover all well-known IBVs (including Insyde, AMI, and Phoenix) and major X86 CPU architectures (e.g., Intel Core, Intel Xeon, Intel Atom, Intel Celeron, Intel Pentium, and AMD Ryzen). We also observed these firmware images have been implemented on top of two widely used open source releases of the UEFI specification [12], EDK II [34] and Project Mu. Project Mu is an open-source release of the UEFI specification that has been leveraged by Microsoft Surface [35].

Finally, we have extracted a total of 28449 SMI handlers from these UEFI firmware images. As is shown in Table I, we claim that this dataset is of high diversity and sufficient scale.

B. Effectiveness

We evaluate the effectiveness of SPENDER by demonstrating its ability in discovering new SMM privilege escalation vulnerabilities in a wide range of UEFI firmware images.

Table I contains the main results. SPENDER has detected 40 potential vulnerabilities from the 1148 stripped COTS UEFI firmware images in our dataset. Each warning report produced by SPENDER consists of one potentially vulnerable SMI handler, an escaped reference and a security-sensitive instruction. Two warnings are not distinguished if they are generated from two different SMM handlers but with the same GUID and the same escaped reference (i.e., the same vulnerability source). After responsible disclosure, 36 vulnerabilities are confirmed by the vendors, including 28 zero-day vulnerabilities with CVE IDs, seven zero-day vulnerabilities pending fixes, and two bug collisions (for the two 1-day vulnerabilities). Of the 34 0-day vulnerabilities found, six reside in Microsoft Surface's UEFI firmware images developed based on Project Mu. For all the confirmed vulnerabilities with assigned CVSSv3 scores,

Table I: The UEFI firmware dataset and main results. Columns 2-5 give the number of images from each of the eight OEMs distributed over three major IBVs. Column 6 gives the number of SMI handlers for each OEM. Columns 7-8 provide the analysis time statistics spent by SPENDER. Column 9 gives the number of reports generated by SPENDER. Columns 10-11 give the number of confirmed vulnerabilities (the number of confirmed vulnerabilities that can be detected when the inter-driver call graph edges are ignored) and their average CVSSv3 scores. All these 36 vulnerabilities have been confirmed as privilege escalation vulnerabilities by the vendors, with the two 1-day vulnerabilities as being known previously, but the 34 0-day vulnerabilities as new (of which 28 have been repaired with CVE IDs credited and the remaining 6 are being repaired).

Characteristics of UEFI Firmware Images						Detection Cost		Vulnerabilities		
OEM	#Images	AMI	Insyde	Phoenix	#SMI Handlers	Total	Per Handler (AVG)	#Reports	#Vuls. (Detectable without Inter-Driver Call Graph Edges)	CVSSv3 Base Score (AVG)
Lenovo	182	4	41	137	5609	16h34m	10.61s	10	8 (1)	7.5
HP	322	313	9	0	8243	30h49m	13.45s	8	7 (3)	7.6
Asus	419	415	0	4	8812	32h08m	13.31s	2	2 (0)	-
Dell	94	31	49	14	2787	6h56m	8.93s	7	7 (2)	6.8
Huawei	4	0	4	0	109	23m	12.66s	6	5 (1)	-
Intel	37	37	0	0	760	4h5m	19.33s	1	1 (1)	7.5
MSI	86	0	0	86	2078	7h22m	12.75s	0	0 (0)	-
Microsoft	4	4	0	0	51	17m	19.98s	6	6 (4)	-
Total	1148	804	103	241	28449	98h37m		40	36 (12)	
Average							12.46s			7.3

Table II: The 28 CVE IDs found by SPENDER (ACE for arbitrary code execution and AAW for arbitrary address write).

CVE ID	CVSS Score(v3)	OEM	CVE State	Escaped Reference	Security Impact
CVE-2020-5348	6.8	Dell	Assigned	Legacy	ACE
CVE-2020-5376	6.8	Dell	Assigned	Legacy	ACE
CVE-2020-5378	6.8	Dell	Assigned	Legacy	ACE
CVE-2020-5379	6.8	Dell	Assigned	Legacy	ACE
CVE-2020-26186	6.8	Dell	Assigned	Legacy	ACE
CVE-2019-16284	7.6	HP	Assigned	Legacy	ACE
CVE-2020-6914	7.6	HP	Reserved	Legacy	ACE
CVE-2020-6915	7.6	HP	Reserved	Legacy	ACE
CVE-2020-5951	-	Insyde	Reserved	Legacy	ACE
CVE-2020-5952	-	Insyde	Reserved	Unintentional	AAW
CVE-2020-5953	-	Insyde	Reserved	Legacy	ACE
CVE-2020-5954	-	Insyde	Reserved	Unintentional	AAW
CVE-2020-5955	-	Insyde	Reserved	Unintentional	AAW
CVE-2020-5956	-	Insyde	Reserved	Unintentional	AAW
CVE-2020-8993	-	Insyde	Reserved	Legacy	ACE
CVE-2020-9284	-	Insyde	Reserved	Unintentional	AAW
CVE-2020-9532	-	Insyde	Reserved	Legacy	ACE
CVE-2020-12139	-	Insyde	Reserved	Legacy	ACE
CVE-2019-6170	9.8	Lenovo	Assigned	Legacy	ACE
CVE-2019-6172	9.8	Lenovo	Assigned	Unintentional	AAW
CVE-2020-8321	6.7	Lenovo	Assigned	Unintentional	AAW
CVE-2020-8322	6.7	Lenovo	Assigned	Unintentional	AAW
CVE-2020-8323	6.7	Lenovo	Assigned	Unintentional	AAW
CVE-2020-8332	6.4	Lenovo	Assigned	Legacy	ACE
CVE-2020-8333	7.8	Lenovo	Assigned	Legacy	ACE
CVE-2020-8354	6.7	Lenovo	Assigned	Unintentional	AAW
CVE-2020-12337	7.5	Intel	Assigned	Legacy	ACE
CVE-2021-36325	7.5	Dell	Assigned	Legacy	ACE

Table III: Statistics for Phase I "Protocol Binding" and Phase II "Inter-Driver Call Graph Construction" in SPENDER.

OEM	Phase I		Phase II	
	GUID	#Protocol Bindings	#Inter-Driver Call Edges	#Call Graph Edges
Lenovo	5719216	281321	353620 (6.42%)	5506429
HP	32660781	178736	290489 (5.72%)	5076686
Asus	10450504	45618	100278 (2.18%)	4586652
Dell	3883958	116609	204171 (10.5%)	1935643
Huawei	136150	13309	17315 (8.53%)	202967
Intel	965999	19677	31838 (5.39%)	590104
MSI	8857065	85452	165165 (9.24%)	1786309
Microsoft	102479	3098	4810 (6.39%)	75239
Total	62776152	743820	1167686 (5.91%)	19760029

the average is 7.3 (higher than 7.0), indicating most of these vulnerabilities are of high-severity.

Table II lists all the 28 vulnerabilities with the CVE IDs credited to us, including their CVSSv3 scores, and whether a vulnerability is caused by a legacy or unintentional reference and its security impact. These are high-severity vulnerabilities, indicated by the high CVSSv3 scores (if assigned already). Their security impact is very high, with 18 of the 28 vulnerabilities exploitable for arbitrary code execution (ACE) and the remaining 10 for arbitrary address write (AAW).

As a protocol-centric static analysis, SPENDER consists of three phases, protocol binding, inter-driver call graph construction, and vulnerability detection. We now analyze how these three phases contribute to its overall effectiveness, with a particular emphasis on the significance of the first two phases working together to discover precisely the inter-driver call graph edges for the protocol-related indirect calls in SMM drivers. As shown in Column 10 in Table I, SPENDER will be significantly less effective if it ignores all the inter-driver call graph edges introduced by SPENDER, by finding only 12 among all the 36 vulnerabilities reported.

Table III provides some statistics about the first two phases of SPENDER for our dataset. During its first phase, SPENDER has identified a total of 743,820 protocol bindings via a total of 62,776,152 unique GUIDs. This indicates that protocols are used pervasively in developing SMM drivers and cannot thus be ignored by any static analysis. During the second phase, SPENDER discovers a total of 1,167,686 inter-driver call graph edges by leveraging these protocol bindings. Without these inter-driver call graph edges (for relating producer and consumer protocols), SPENDER can only find 12 among the 36 vulnerabilities reported as mentioned above.

Table IV gives some statistics about the last phase of SPENDER (in finding initially all the escaped references in its Step 1 and identifying subsequently the unsafe ones (i.e., the ones leading to security-sensitive instructions without sanity checks) in its Step 2). In Step 1, SPENDER has identified a total of 108,586 escaped references, consisting of 64,445 legacy references and 44,141 unintentional references (with 28,449 to *CommBuffer* and 15,692 to *SavedState*). In Step 2, SPENDER has found a total of 36 unsafe escaped

Table IV: Statistics for Phase III "Vulnerability Detection" in terms of the number of escaped (unsafe) references found.

OEM	#Legacy References	#Unintentional References	
		CommBuffer	SavedState
Lenovo	7846 (3)	5609 (0)	4443 (5)
HP	22892 (3)	8243 (0)	4518 (4)
Asus	20412 (1)	8812 (0)	2757 (1)
Dell	5412 (3)	2787 (0)	1396 (4)
Huawei	164 (1)	109 (0)	28 (4)
Intel	1940 (1)	760 (0)	324 (0)
MSI	5689 (0)	2078 (0)	2214 (0)
Microsoft	90 (3)	51 (1)	12 (2)
Total	64445 (15)	28449 (1)	15692 (20)

```
1. typedef void (*fPtr)();
2. struct Protocol1 {
3.   fPtr * func;
4. } protocol;

5. EFI_GUID guid = "F51DD33A-E57F-4020-B466-F4C171C6E4F7";

6. void func(){
7.    ...
8. }

9. EFI_BOOT_SERVICES* BootServices;

10. void Init_func(EFI_HANDLE HI, EFI_SYSTEM_TABLE* ST)
11.{
12.    BootServices = ST->BootServices.
13.    protocol.func = &func;
14.    InstallProtocol(GUID, &protocol);
15. }
```

An SMM Driver (Producer)

```
16. struct Protocol1 * protocolPtr;
17. EFI_GUID guid = "F51DD33A-E57F-4020-B466-F4C171C6E4F7";
18. void init_func(EFI_HANDLE HI, EFI_SYSTEM_TABLE* ST){
19.    Bs = ST->BootServices;
20. }

21. void __fastcall SMI_handler()
22. {
23.    Bs->LocateProtocol(GUID, &protocolPtr);// Vulnerability Occurs
24.    protocolPtr->func();
25. }
```

An SMM Driver (Consumer)

Figure 7: **CVE-2020-5376**

references (leading to the 36 vulnerabilities reported). The remaining escaped references that are accessed in some SMI handlers are safe for two reasons: (1) the SMI handlers perform sanity checks against the escaped references operated on ([PRINCIPLE-II]), and (2) the SMI handlers do not use security-sensitive instructions ([PRINCIPLE-I]).

Finally, we examine two representative SMM drivers (with reverse-engineered code snippets) that contain one SMM privilege escalation vulnerability each, detected by SPENDER.

CVE-2020-5376. This vulnerability exists in an SMM driver that appears in multiple Dell UEFI firmware images. It can be exploited to execute arbitrary code in system management mode (SMM). Figure 7 gives its vulnerability-relevant code snippet.

To detect this vulnerability, SPENDER first binds protocolPtr in the consumer to protocol in the producer. By resolving the indirect call at Line 24, SPENDER discovers an inter-driver call graph edge from the instruction at Line 24 to the protocol function, func, declared at Line 6. At Line 23, a function, which is pointed by a field of a global variable, BootServices, is called. As BootServices is a legacy reference, its content is controllable by the kernel at ring 0, leading

```
1. void SMI_handler(){
2.   ...
3.   int * v1;
4.   // Get the value of rbx in CPU save state to v1
5.   SMMCpuProtocol->ReadSaveState(SMMCpuProtocol, 4, rbx_index, 0, &v1);
6.   ...
7.   v1[1] = 0;
8. }
```

Figure 8: CVE-2020-5955.

to a vulnerability that can cause arbitrary code execution at Line 23.

CVE-2020-5955. This vulnerability exists in an SMM driver, named Int15MicrocodeSmm, that appears in multiple Insyde UEFI firmware images. It can be exploited to perform arbitrary write to the SMRAM (a privileged memory region). Figure 8 gives its vulnerability-relevant code snippet.

To detect this vulnerability, SPENDER finds that v1 is an unintentional reference as it points to the value of rbx stored in SavedState at Line 5. Obviously, the value of v1 is controllable by the kernel at ring 0, as the value of rbx can be set by the kernel before this particular SMI handler is triggered. If the value of v1 is used yet not sanitized, as shown at Line 7, a vulnerability that can cause arbitrary memory write arises.

C. Precision

SPENDER generates a total of 40 vulnerability warning reports (Table I). By analyzing these reports manually, in collaboration with all the relevant vendors, we find only four false positives. This gives rise to a false positive rate of 10%. For the four false positives, we find that they all occur when their vulnerable SMI handlers can never be executed at run time. There are two cases: (1) a certain SMM driver has unregistered a vulnerable SMI handler that was registered earlier during the boot phase, and (2) a certain SMM driver has replaced a vulnerable SMI handler with a secure one with the same GUID so that only the secure one is used.

To evaluate SPENDER to understand its false negative rate, we build a dataset with known vulnerabilities as the ground truth. To gather SMM-related vulnerabilities, we collect all the security bulletins from the official security update websites and restrict our search to SMM privilege escalation vulnerabilities using such keywords as "SMM", "SMI" and "System Management Mode". This gives us a total of 10 vulnerabilities from eight UEFI firmware images. We then manually analyze the vulnerable and patched UEFI firmware binaries crawled from the official security update websites, locating these 10 vulnerabilities given in Table V. By applying SPENDER to these eight images, we obtain a total of 15 vulnerability warning reports, consisting of not only these 10 previously known vulnerabilities but also five new ones.

D. Efficiency

As is shown in Table I, SPENDER is able to analyze all the 1148 UEFI firmware images within five days. Table VI gives a breakdown of SPENDER's analysis time across its three phases. On average, its first phase is the most time-consuming due to the time taken in finding data and protocol dependencies, building CFGs, and constructing call graphs. However, SPENDER is still efficient, by spending only an average of 12.46 seconds on analyzing a single SMI handler.

Table V: SPENDER's ability in detecting both previously known and unknown SMM privilege escalation vulnerabilities (abbreviated to KVs and UKVs, respectively).

Dataset of Known Vulnerabilities			Vulnerabilities Detected	
Firmware (Computer System)	Vulnerability ID	Escaped Reference	#KVs	#UKVs
IdeaCentre 310S	LEN-30042-AMI	Unintentional	1	0
HP 245 G7	CVE-2020-12890	Unintentional	1	0
Thinkpad T480	CVE-2019-6170 CVE-2019-6172	Unintentional Legacy	2	1
ThinkCentre M700	CVE-2017-3753	Unintentional	1	2
ThinkSystem SD650	LEN-24238	Unintentional	1	0
Inspiron 5378	CVE-2019-0185	Unintentional	1	2
NUC7i7DNHE	CVE-2019-11163	Unintentional	1	0
W281-G40	CVE-2019-11136 CVE-2019-11137	Unintentional Unintentional	2	0
Total	10		10	5

Table VI: A breakdown of SPENDER's detection cost.

OEM	Detection Cost Per Handler On Average (secs)			
	Phase I	Phase II	Phase III	Total
Lenovo	5.02 (48%)	3.81 (35%)	1.78 (17%)	10.61 (100%)
HP	6.70 (49%)	4.45 (33%)	2.30 (18%)	13.45 (100%)
Asus	6.75 (51%)	4.23 (32%)	2.13 (17%)	13.31 (100%)
Dell	4.21 (47%)	3.20 (35%)	1.52 (18%)	8.93 (100%)
Huawei	6.61 (52%)	3.85 (31%)	2.20 (17%)	12.66 (100%)
Intel	8.36 (45%)	7.42 (37%)	3.55 (18%)	19.33 (100%)
MSI	6.75 (52%)	4.21 (33%)	1.79 (15%)	12.75 (100%)
Microsoft	8.88 (50%)	7.35 (34%)	3.75 (16%)	19.98 (100%)
Total	6.25 (50%)	4.17 (33%)	2.04 (17%)	12.46 (100%)

E. Discussions

1) Protocol-Centric Static Analysis: By adopting a protocol-centric analysis, SPENDER represents an effective static framework for detecting SMM privilege escalation vulnerabilities in stripped COTS UEFI firmware images efficiently and precisely. By relating consumer and producer protocols via their commonly shared GUIDs in its first phase, SPENDER can discover precisely all the inter-driver call graph edges in a path-sensitive manner in its second phase. As a result, SPENDER can sharpen its flow- and context-sensitive vulnerability detection in its last phase both efficiently and precisely. In contrast, existing static analyses for stripped binaries will be either highly unsound [15], [16], [17], [18] or highly imprecise [13], [14], [24], [25], [26] when applied to discover indirect call targets at the protocol-related indirect call instructions in UEFI firmware images (as further discussed in Section VI).

The basic idea behind our protocol-centric analysis can also be applied to programs written in other callback-based mechanisms such as windows RPC [19] and COM [20].

2) Security Impact of SMM Privilege Escalation Vulnerabilities: All the 36 vulnerabilities found by SPENDER are of high severity. For all the CVSSv3 scores assigned, their average is 7.3 (Table I), which may not seem to be severe enough at the first glance. However, we argue that the actual security threats posed by these vulnerabilities are underestimated unduly by these CVSSv3 scores, which are assigned based on the assumption that acquiring the root privilege of a computer system makes it hard to exploit these vulnerabilities, even though all of them can lead to privilege escalation attacks, as demonstrated by us to the relevant vendors.

SPENDER does not regard indirect memory read instructions as being security-sensitive. While such read instructions may be used to access arbitrary memory addresses in the SMRAM potentially, the contents read illegally can only be leaked out of the SMRAM via an arbitrary memory write instruction. However, SPENDER has already treated indirect memory write instructions as being security-sensitive.

3) Mitigation Techniques: There can be two approaches to mitigating the SMM privilege escalation vulnerabilities in UEFI firmware. One is to provide a standard programming interface for checking the validity of data coming from outside the SMRAM. However, the input validation that would be required will be a common troublesome problem, which belongs to the top 25 most dangerous weaknesses in 2020 [36], since a sound sanity check should enforce the correctness of its value in a specific execution context (e.g., by requiring a pointer to fall within an expected address range). For an SMI handler, a sound sanity check should ensure that every memory access made by an SMI handler lies within the address space of the SMRAM. Currently, how to ensure that a correct sanity check is always used where needed remains unsolved.

The other approach is to deploy hardware-assisted mitigation mechanisms. To restrict the range of memory addresses accessed, Supervisor Mode Access Prevention (SMAP) or Intel's STM [10] can be considered. However, as mentioned in Intel's 2020 white paper [5], "most current BIOS's do not support the STM or other SMM monitors." In this research, we have not found a single UEFI firmware image that supports STM or other STM monitors. In the case of SMM drivers, other alternatives, such as SMM Code Access Check [6], [7], [8] (much like Supervisor Mode Execution Prevention (SMEP)) and DEP/NX [5], have been deployed but are all bypassable by exploiting jump-oriented programming [9]. How to secure legacy firmware effectively is still challenging.

Finally, severe security risks in UEFI firmware are sometimes attributed to the increasing complexity in SMM drivers shipped along a somewhat long supply chain. For example, an upstream vulnerable SMM driver (e.g., provided by some IBVs) is usually directly integrated into UEFI firmware images by downstream OEMs. To reduce or even eliminate security risks, downstream OEMs should not only develop their own SMM drivers strictly according to [PRINCIPLE-I] and [PRINCIPLE-II] defined in the UEFI specification [12] but also treat all upstream SMM drivers with caution.

4) Improvements and Limitations: We can further sharpen the precision of SPENDER by modeling the data structures for protocols in SMM drivers more precisely. This is the same challenge faced by all static analyses for stripped binaries. We can also further improve the efficiency of SPENDER by adopting a demand-driven approach to cutting down the analysis time spent during its protocol binding phase (Table VI).

SPENDER is limited by the availability of UEFI firmware images, especially incompletely updated firmware images supplied by the vendors. In addition, SPENDER focuses

on detecting SMM privilege escalation vulnerabilities only and are thus not capable of detecting many other kinds of vulnerabilities (e.g., incorrect authorization).

VI. Related work

Understanding SMM Privilege Escalation Vulnerabilities. Several recent works [6], [7], [8] have investigated how to exploit SMM privilege escalation vulnerabilities to hijack an SMI handler's control flow and execute arbitrary code with SMM privilege. However, these vulnerabilities have all been found manually, and their root causes have not been studied systematically. In this paper, we present a systematic study on such vulnerabilities and identify their root causes as being two types of references that may escape from the SMRAM. There are other attacks against SMM (e.g., cache-poisoning [22]) that are designed also to achieve privilege escalation. However, this paper does not focus on detecting such attacks.

Detecting SMM Privilege Escalation Vulnerabilities. There are many attempts on finding privilege escalation vulnerabilities in the operating system kernel components. In particular, kAFL [37], Syzkaller [38] and TriforceAFL [39] can expose a range of kernel bugs by performing coverage-guided kernel fuzzing. In the case of detecting SMM privilege escalation vulnerabilities, we are aware of only one earlier attempt [11], which resorts to a dynamic analysis by combining symbolic execution, fuzzing and concrete testing. However, such a dynamic analysis can hardly be effective, as it will be difficulty to achieve good code coverage by emulating a UEFI environment that is heavily hardware-dependent and applying large-scale fuzz testing to a wide range of UEFI firmware images. In this paper, we introduce SPENDER, the first static analysis that can detect a large number of new SMM privilege escalation vulnerabilities in a wide range of stripped COTS UEFI firmware images efficiently and precisely.

Static Analyses for Stripped Binaries. The problem of finding indirect call targets statically in stripped binaries is challenging. In the case of a UEFI firmware image, where its SMM drivers manage cross-driver communication via protocols identified by GUIDs, any static analysis that can identify effectively the indirect call targets (representing protocol functions) at protocol-related indirect call instructions must be path-sensitive. As a result, existing general-purpose static analyses for stripped binaries are either highly unsound [15], [16], [17], [18] or highly imprecise [13], [14], [24], [25], [26].

Binary analysis tools, such as *IDA Pro* [15] and *Angr* [16], [17], [18], on which SPENDER is built, are known to be highly unsound by often missing many indirect call targets. In addition, by using these two tools alone at the beginning of this research, we did not manage to find any SMM privilege escalation vulnerability in our image dataset. By applying Phases I and III but ignoring Phase II (i.e., the inter-driver call graph edges discovered in Phase II), SPENDER can find only 12 among the 36 vulnerabilities reported (Table I).

Existing static analyses [24], [25], [26] introduced for enforcing control-flow integrity at the binary level are highly imprecise in identifying indirect call targets, since they must over-approximate the set of targets at an indirect call instruction in order to ensure program correctness. Some other general-purpose static analyses for binaries [13], [14] are also highly imprecise (due to the lack of path-sensitivity). For the object-reaching definition analysis introduced earlier in [13] for resolving indirect call targets in C++ binaries, its standard data-flow-based analysis is inherently path-insensitive. For the block-based pointer analysis introduced recently in [14] for analyzing C binaries, its algorithm is also path-insensitive since it does not track the offsets in each memory block.

The multi-layer type analysis [40], which exploits type hierarchies to reduce false indirect call targets in C/C++ binaries, is also ineffective in our setting, since protocols are installed into and retrieved from a protocol mapping table path-sensitively (in terms of void pointers) (Figure 1). Tiara [41] focuses on recovering the container type of a given address in stripped COTS C++ binaries.

Given a UEFI firmware image, we handle its call graph construction and vulnerability detection separately. We first exploit protocol bindings to build a precise call graph for the SMM drivers in the image path-sensitively (Phases I and II). We then perform vulnerability detection in these SMM drivers flow- and context-sensitively as a taint analysis (Phase III).

Taint Analysis. Taint analysis is widely used for detecting bugs/vulnerabilities in firmware binaries [30], [42], Android programs [43], [44], [45], [46], [47] and Linux programs [48]. By identify escaped references as taint sources, our value-flow analysis (i.e., (EspRefs_Driven_Detection() in Algorithm 3) performs essentially a flow- and context-sensitive taint analysis on a pre-computed call graph for SMM drivers.

VII. Conclusion

In this paper, we present a systematical study of SMM privilege escalation vulnerabilities in UEFI firmware. We identify the root causes for such vulnerabilities as two types of escaped references (i.e., the references that escape from the SMRAM), which may flow to security-sensitive instructions without prior sanity checks, thereby enabling a malicious kernel-space program at ring 0 to escalate its privilege to ring -2. We introduce, SPENDER, an effective static protocol-centric analysis framework for uncovering SMM privilege escalation vulnerabilities efficiently and precisely. SPENDER has discovered a total of 36 vulnerabilities (consisting of two 1-day vulnerabilities known previously and 34 new 0-day vulnerabilities) in a total of 1148 stripped COTS UEFI firmware images at a false positive rate of 10% by analyzing each SMI handler in an average of 12.46 seconds.

VIII. Acknowledgments

We thank reviewers for their constructive comments. This work is supported in part by Chinese National Natural Science Foundation (61802394, U1836209, 62032010, 62102411), National Key Research and Development Program of China (2016QY071405), Strategic Priority Research Program of the CAS (XDC02040100, XDC02030200, XDC02020200), and ARC Grants DP180104069 and DP210102409.

REFERENCES

[1] statista, "Total unit shipments of personal computers (PCs) worldwide from 2006 to 2020," 2021, https://www.statista.com/statistics/273495/global-shipments-of-personal-computers-since-2006/.

[2] D. Oleksiuk, "Exploiting ami aptio firmware on example of intel nuc," http://blog.cr4.sh/2016/10/exploiting-ami-aptio-firmware.html, 2016.

[3] D. Oleksiuk", "Thinkpwn," https://github.com/Cr4sh/ThinkPwn, 2017.

[4] HP, "The lojax attack: What you need to know," https://press.ext.hp.com/us/en/blogs/2018/the-lojax-attack--what-you-need-to-know.html?jumpid=in_r12129_us/en/psg/computer_security/sure-start/blog, 2018.

[5] L. Jarlstrom, "A tour beyond bios secure smm communication in the efi developer kit ii," https://github.com/tianocore/tianocore.github.io/wiki/EDK-II-white-papers, 2020.

[6] Bruno, "Code check (mate) in smm," https://www.synacktiv.com/posts/exploit/code-checkmate-in-smm.html, 2018.

[7] Bruno, "Through the smm-class and a vulnerability found there," https://www.synacktiv.com/posts/exploit/through-the-smm-class-and-a-vulnerability-found-there.html, 2020.

[8] S. E. Lab, "Smm unchecked pointer vulnerability," http://esec-lab.sogeti.com/posts/2016/05/30/smm-unchecked-pointer-vulnerability.html, 2016.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 30–40. [Online]. Available: https://doi.org/10.1145/1966913.1966919

[10] E. D. Myers, "Using the intel stm for protected execution," 2018.

[11] J. Engblom, "Finding bios vulnerabilities with symbolic execution and virtual platforms," https://software.intel.com/en-us/articles/finding-bios-vulnerabilities-with-symbolic-execution-and-virtual-platforms, 2017.

[12] uefi.org, https://uefi.org/sites/de-fault/files/resources/UEFI_Spec_2_8_final.pdf, 2019.

[13] D. Dewey and J. T. Giffin, "Static detection of c++ vtable escape vulnerabilities in binary code." in *NDSS*, 2012.

[14] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level," in *Network and Distributed System Security Symposium (NDSS)*, 2021.

[15] Hex-Rays, "Hex-rays ida pro: A powerful disassembler and a versatile debugger," https://www.hex-rays.com/ida-pro/, 2021.

[16] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[17] F. Gritti, L. Fontana, E. Gustafson, F. Pagani, A. Continella, C. Kruegel, and G. Vigna, "Symbion: Interleaving symbolic with concrete execution," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, June 2020.

[18] R. Wang, Y. Shoshitaishvili, A. Bianchi, M. Aravind, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[19] Microsoft, "Remote Procedure Call," 2021, https://docs.microsoft.com/en-us/windows/win32/rpc/registering-interfaces.

[20] Wiki, "Component Object Model," 2021, https://en.wikipedia.org/wiki/Component_Object_Model.

[21] uefi.org, https://uefi.org/sites/default/files/resources/PI_Spec_1_7_final_Jan_2019.pdf, 2019.

[22] R. Wojtczuk and J. Rutkowska, "Attacking smm memory via intel cpu cache poisoning," *Invisible Things Lab*, pp. 16–18, 2009.

[23] Intel, "Intel hardware shield: Trustworthy smm on the intel vpro platform," https://www.intel.com/content/dam/www/central-libraries/us/en/documents/drtm-based-computing-whitepaper.pdf, 2020.

[24] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. USA: IEEE Computer Society, 2013, p. 559–573. [Online]. Available: https://doi.org/10.1109/SP.2013.44

[25] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[26] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 927–940.

[27] C. Kallenberg, J. Butterworth, X. Kovah, and C. Cornwell, "Defeating signed bios enforcement," *EkoParty, Buenos Aires*, 2013.

[28] L. Zhang, J. Chen, W. Diao, S. Guo, J. Weng, and K. Zhang, "Cryptorex: Large-scale analysis of cryptographic misuse in iot devices," in *RAID*, 2019.

[29] N. W. Group, "Rfc 4122," https://datatracker.ietf.org/doc/html/rfc4122.

[30] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: Detecting the taint-style vulnerability in embedded device firmware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 430–441.

[31] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1769–1786. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lu

[32] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in os kernels," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1899–1913.

[33] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 49–61.

[34] Intel, "Edk ii project," https://github.com/tianocore/edk2, 2021.

[35] Microsoft, "Introducing project mu," https://blogs.windows.com/windowsdeveloper/2018/12/19/%E2%80%AFintroducing-project-mu/, 2018.

[36] CWE, "2020 cwe top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html, 2020.

[37] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for OS kernels," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[38] D. Drysdale, "Coverage-guided kernel fuzzing with syzkaller," *Linux Weekly News*, vol. 2, p. 33, 2016.

[39] J. Hertz and T. Newsham, "Triforceafl," 2017.

[40] K. Lu and H. Hu, "Where does it go?: Refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 1867–1881.

[41] X. Wang, X. Xu, Q. Li, M. Yuan, and J. Xue, "Recovering container class types in binaries," ser. CGO '22, 2022.

[42] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1544–1561.

[43] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3183575

[44] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[45] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1007–1024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

[46] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "FIRMSCOPE: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2379–2396. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/elsabagh

[47] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue, "Performance-boosting sparsification of the ifds algorithm with applications to taint analysis," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 267–279. [Online]. Available: https://doi.org/10.1109/ASE.2019.00034

[48] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in linux device drivers," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 255–268.