EfiDrill Write Paper

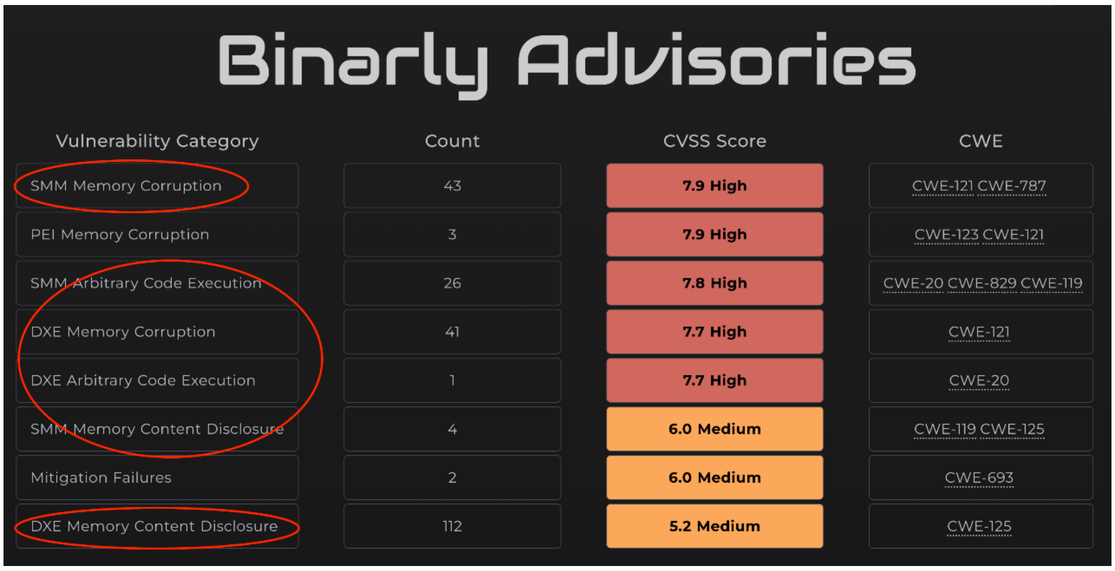
1. Abstract

This white paper aims to serve as a supporting and reference material for the presentation “EfiDrill ——Automated Hunting UEFI Firmware Vulnerability through Data-Flow Analysis” at Black Hat EU 2023. The presentation and this paper focus on our newly developed automated UEFI firmware vulnerability discovery tool, EfiDrill, discussing the limitations of existing open-source tools and how EfiDrill extends detection capabilities based on those tools, which encompass real 0-day vulnerability analysis and implementation details of EfiDrill.

2. Content Primer

Amidst ongoing research into UEFI security, researchers have discovered numerous SMM vulnerabilities, enhancing the robustness of UEFI. Remarkably, the emergence of tools like “efiXplorer” has significantly streamlined the reverse engineering process for UEFI firmware.

According to the findings from Binary-IO’s security bulletin, a majority of vulnerabilities exist in SMM drivers and DXE drivers, with a noteworthy concentration of high-risk vulnerabilities identified in the former.



Binarly Advisories

Vulnerability Category	Count	CVSS Score	CWE
SMM Memory Corruption	43	7.9 High	CWE-121 CWE-787
PEI Memory Corruption	3	7.9 High	CWE-123 CWE-121
SMM Arbitrary Code Execution	26	7.8 High	CWE-20 CWE-829 CWE-119
DXE Memory Corruption	41	7.7 High	CWE-121
DXE Arbitrary Code Execution	1	7.7 High	CWE-20
SMM Memory Content Disclosure	4	6.0 Medium	CWE-119 CWE-125
Mitigation Failures	2	6.0 Medium	CWE-693
DXE Memory Content Disclosure	112	5.2 Medium	CWE-125

Figure 1

System Management Mode (SMM) is a CPU execution mode introduced by Intel for the x86 architecture. Access to SMM can only be achieved through System

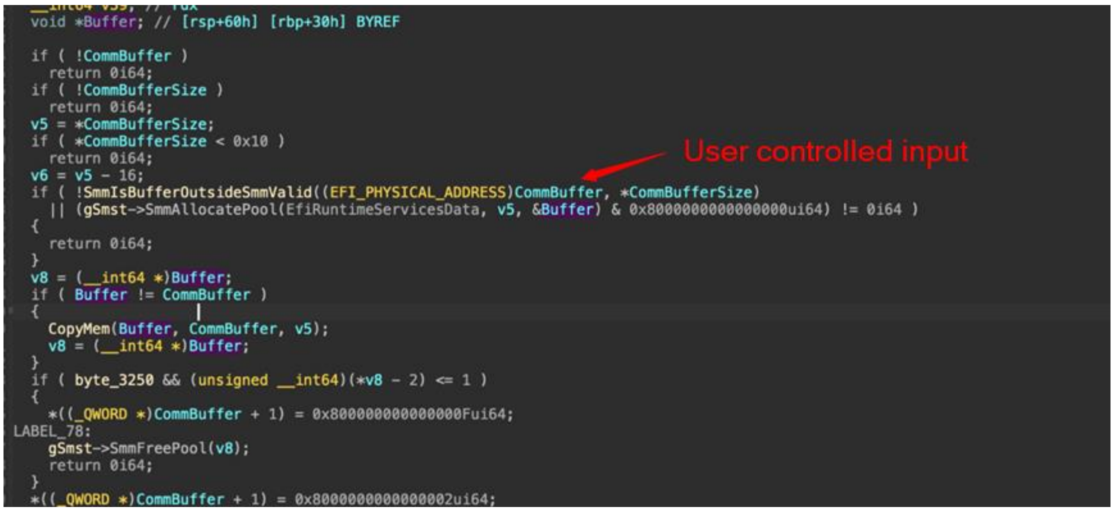
Management Interrupts (SMI) and can only be exited by executing RSM instruction. SMM remains invisible to the operating system (OS) and is referred to as ring-2. If malicious actors are to gain control of SMM, they can bypass Secure Boot and obtain BitLocker keys or implant a rootkit undetectable to the operating system.

SMM's attack surface includes:

- CommBuffer: Allows carrying external data into the Child SMI using the corresponding GUID.
- Ports and data requested by SMI.
- 0x40E: An SMM external address, which may be used even in legacy BIOS, despite potential security risks.
- Getvariable: A function for variable retrieval, through which the data acquired may be subject to modification by the operating system, thus resulting in potential risks.
- Runtime Service Interfaces: which are stored outside of SMM. Calling them in SMM will trigger a Callout.
- ReadSaveState: Allows retrieval of the register value when calling SMI; however, the trustworthiness of the register's value itself is not guaranteed.

3. Motivations

Our team, Lenovo Global Security Lab (China), is responsible for conducting security review on Lenovo's internal UEFI products, boasting a wealth of expertise garnered over years of assiduously exploring UEFI security vulnerabilities. In 2022, we tracked a security advisory released by Intel, bearing the appellation intel-sa-00688, delineating a high-risk vulnerability, its CVSS score soaring to an alarming 7.9, identified as CVE-2022-21198. Regrettably, the security advisory provided limited details, only mentioning to its essence as a TOCTOU vulnerability. This vulnerability caught our attention, and through reverse engineering, we identified the corresponding SMM driver - SpiSmmStub, and located the specific vulnerability.



```
  void *Buffer; // [rsp+60h] [rbp+30h] BYREF

  if ( !CommBuffer )
    return 0i64;
  if ( !CommBufferSize )
    return 0i64;
  v5 = *CommBufferSize;
  if ( *CommBufferSize < 0x10 )
    return 0i64;
  v6 = v5 - 16;
  if ( !SmmIsBufferOutsideSmmValid((EFI_PHYSICAL_ADDRESS)CommBuffer, *CommBufferSize)
    || (gSmst->SmmAllocatePool(EfiRuntimeServicesData, v5, &Buffer) & 0x8000000000000000ui64) != 0i64 )
  {
    return 0i64;
  }
  v8 = (__int64 *)Buffer;
  if ( Buffer != CommBuffer )
  {
    CopyMem(Buffer, CommBuffer, v5);
    v8 = (__int64 *)Buffer;
  }
  if ( byte_3250 && (unsigned __int64)(*v8 - 2) <= 1 )
  {
    *((_QWORD *)CommBuffer + 1) = 0x800000000000000Fui64;
LABEL_78:
    gSmst->SmmFreePool(v8);
    return 0i64;
  }
  *((_QWORD *)CommBuffer + 1) = 0x8000000000000002ui64;
```

Figure 2

In the "if" statement shown in the figure 2, the function SmmIsBufferOutsideSmmValid checks whether the external input variable CommBuffer overlaps with SMM. However, it is imperative to highlight that the check assumes that CommBuffer is numerical. Should CommBuffer store a pointer, the function cannot ascertain overlaps with SMM's memory space, thereby introducing inherent security vulnerabilities.

In figure 2, the variable CommBuffer stores a pointer that is under the control of an attacker, which results in the destination address and size of the final CopyMem call being susceptible to manipulation by the attacker.

When an SMI call is triggered, attackers can exploit this vulnerability quite easily through a DMA attack. This is due to the presence of I/O operations within the SMI call, creating a highly susceptible DMA attack window. In retrospect, we believe that conducting data flow analysis on UEFI can enhance the likelihood of identifying and discovering such vulnerabilities.

As research progresses, we have discovered that many UEFI vulnerabilities are deeply concealed, entailing intricate parameter propagation and data interchanges. Many latent UEFI vulnerabilities evade conventional detection techniques, with existing UEFI vulnerability detection tools primarily relying on fuzz testing or assembly instruction matching. Regrettably, no publicly available tool exists that can automatically detect and discover UEFI security vulnerabilities through data flow tracking analysis. Therefore, we developed EfiDrill - the first open-source IDA plugin for data flow analysis of UEFI firmware.

We have open-sourced EfiDrill, a tool that enables data flow tracing, taint tracking, automated structure analysis, variable numerical prediction, and automated vulnerability detection for SMM external data. Within approximately one month, we have discovered dozens of UEFI firmware bugs, with more than ten of them acknowledged as vulnerabilities by renowned vendors. We intend to transform the tool into an IDA plugin and release it as an open-source project, thus enabling a broader community of security researchers to leverage this tool for UEFI vulnerability analysis and assist in enhancing the security of UEFI products.

4. Current UEFI Static Analysis Tools and Challenges

The static analysis represents a prevalent methodology in UEFI vulnerability analysis, offering insights into the intricacies of code logic. It enables the identification of code logic that might remain elusive during fuzz testing, yet hold critical significance. Presently, several remarkable UEFI static analysis tools merit attention:

- IDA: Focused on firmware symbol recovery, IDA streamlines the complexity of reverse engineering.
- efiXplorer: As the pioneer of automated UEFI static analysis, efiXplorer adeptly discovers vulnerabilities such as double Getvariable and runtime call out. Furthermore, it facilitates the identification of specific GUIDs within IDA, enhancing the efficacy of reverse engineering endeavors.
- efi_fuzz: Pioneering the simulation execution of UEFI firmware for fuzz testing, efi_fuzz excels in discovering memory corruption vulnerabilities.
- chipsec: Designed to discern UEFI configuration flaws, chipsec proficiently identifies errors in register configuration.

These tools have already revealed numerous UEFI vulnerabilities, thereby contributing significantly to the security enhancement of UEFI products. However, their effectiveness in analyzing intricate UEFI vulnerabilities encounters notable challenges. To address this, we aspire to introduce sophisticated vulnerability discovery methods, such as data-flow analysis, to discover more complex and deeper vulnerabilities. This endeavor has spurred the creation of our novel vulnerability discovery tool, EfiDrill.

5. Glimpse of EfiDrill

In response to the aforementioned situation of UEFI automated vulnerability discovery, EfiDrill introduces the following novel functionalities:

- Enabling data-flow analysis in UEFI.
- Achieving numerical prediction.
- Realizing automated structure analysis.
- Integrating a plugin system for analysis components, which can help discover multiple 0-day vulnerabilities.

6. Design and Implementation of EfiDrill

6.1 Architecture Design

EfiDrill adheres to a hierarchical design principle, comprising three layers: management and scheduling layer, analysis and processing layer, and detection plugin layer:

- The management and scheduling layer consists of a management module that contains a list of all functions awaiting analysis, along with APIs.
- The analysis and processing layer is responsible for handling specific analytical tasks. It analyzes each function by conducting variable numerical prediction and preserving data-flow relationships. This layer includes plugin manager module and analysis engine module, which comprises data-flow analysis module and the numerical prediction module.
- The detection plugin layer scrutinizes the current function to identify any issues matching predefined rules. This layer encompasses vulnerability point detection plugin module and structure type analysis module.

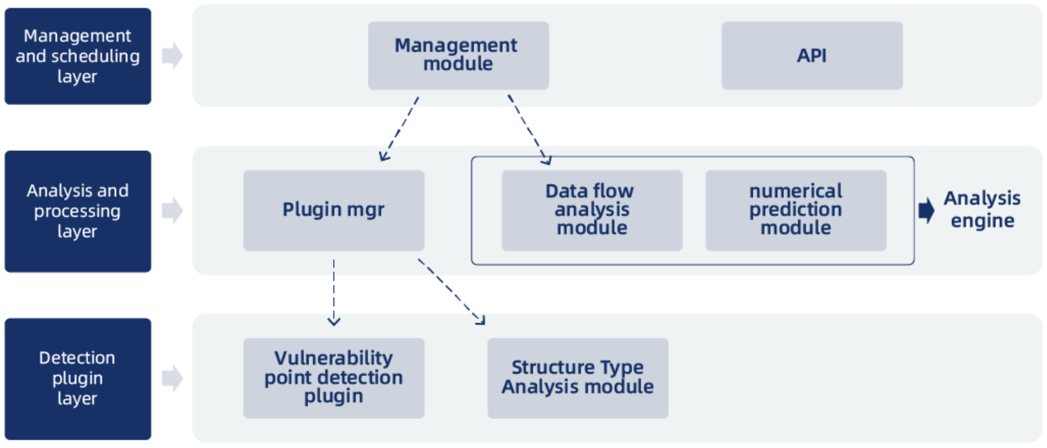

Management and scheduling layer
Management module
API
Analysis and processing layer
Plugin mgr
Data flow analysis module
numerical prediction module
Analysis engine
Detection plugin layer
Vulnerability point detection plugin
Structure Type Analysis module

Figure 3

6.2 Analytical Engine

6.2.1 Data-Flow Analysis Module

a) Analysis of Use-Def Chains for Data

We commence by reverse engineering the firmware and converting the obtained assembly code into intermediate representation (IR) language. Subsequently, we construct the use-def sets for the data through reaching definition analysis and then perform reachability analysis on the addresses. This process is accomplished through the following steps:

- Firstly, we abstract the assembly code of the binary program into IR language on a per-function basis.
- Next, for each function, we employ the reaching definition analysis algorithm to acquire the use-def variables at each address within the function.
- By treating addresses as def variables and conducting reachability analysis, we can determine whether there is a reachable path between two addresses.

To analyze variable propagation across functions, we partition the function into code blocks when function calls occur, and then perform reaching definition analysis.

Figure 4 illustrates the working principle of reaching definition analysis. Representing the assembly code as three-address code (3AC) IR language ensures that each statement defines only one variable. Whenever a new variable is defined, we group the variable with its defining address as a use-def variable, with the format <Var, Var-Addr>.

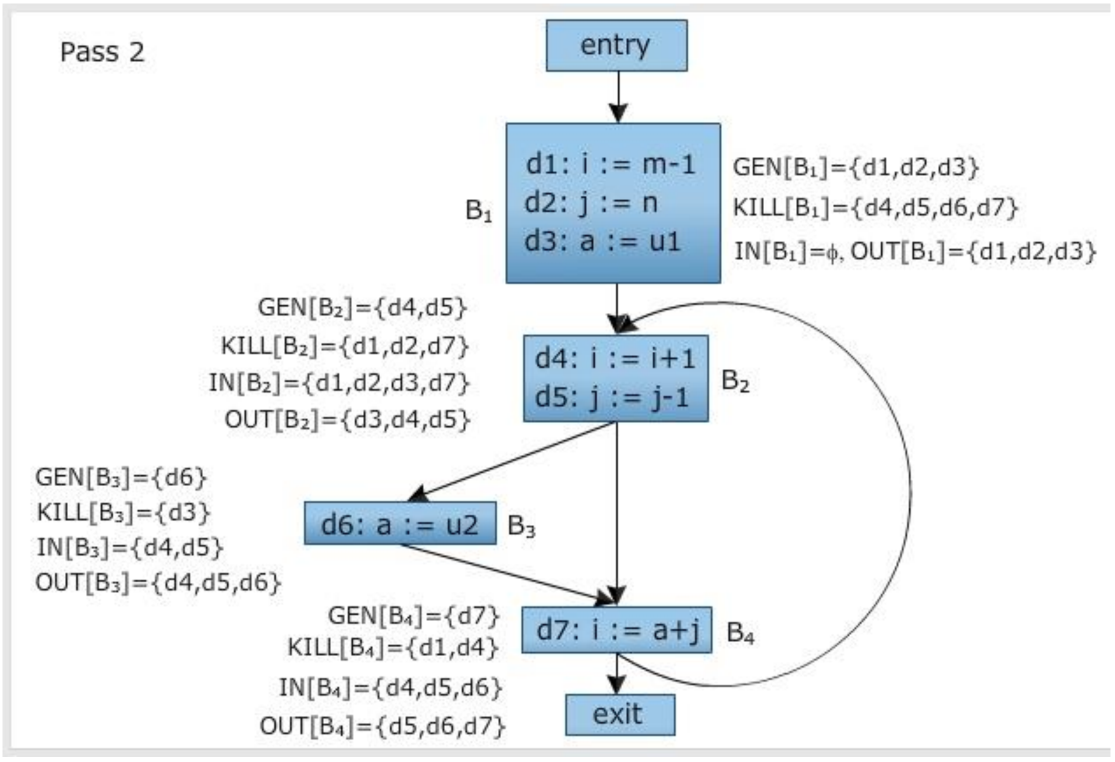


Figure 4

As for function calls, we segment the code on a per-function basis. By doing so, we only need to analyze the propagation paths of use-def variables when function calls occur. In other words, by figuring out the origin of one use-def variable, the entire object's use-def variable propagation paths could be annotated. For instance, as depicted in figure 5, as the entry variable of function F2 originates from function F1's output, we can directly view the analysis result of the entry variable of function F2 as the result of the cross-function variable propagation path, thus avoiding repetitive analyses of functions.

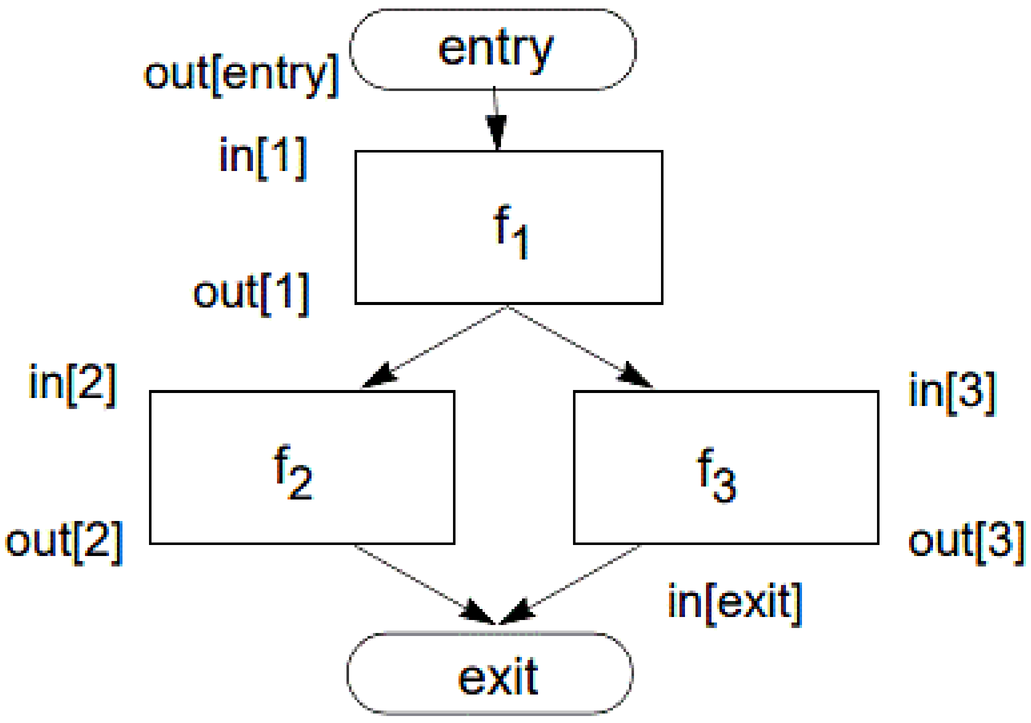


Figure 5

b) Data Flow Tracing

Subsequently, we employ context analysis and alias analysis to mark controllable variables outside of SMM. This process involves the following steps:

- Identifying the initial variables in the specified function that are subject to external control in SMM and placing them at the beginning of our tracing list.
- Analyzing the context to identify the data flow relationships of use-def variables.
- Employing alias analysis algorithms to identify the aliases of certain variables.
- Continuously augmenting the list of controllable variables outside of SMM with newly discovered variables.

Through the aforementioned approach, the data propagation relationships in the Control Flow Graph (CFG) can be seen. As illustrated in figure 6, variable "m" is defined in D3 while variable "k" is used, and in D7, variable "x" is defined and variable "m" is used, which proves the reachability between D7 and D3. Based on this analysis, data flow sequence can be determined: k -> m -> x, indicating that the value of "x" is defined by "k".

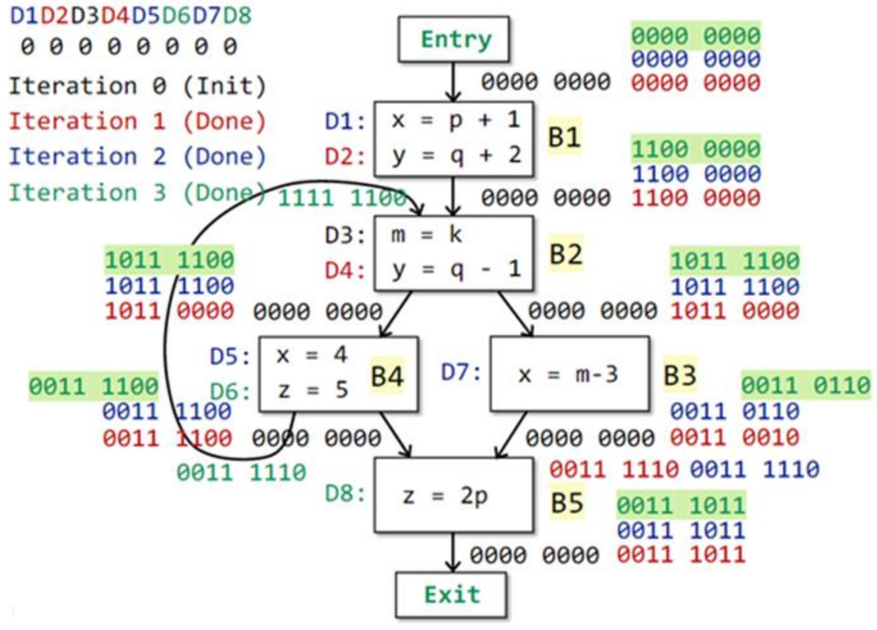


Figure 6

c) Initial Labeling of SMM External Controllable Variables

In order to discover more UEFI firmware vulnerabilities, we have systematically identified all potential attack surfaces originating from the SMM external environment. Following the approach mentioned earlier, we designate all variables corresponding to these SMM external controllable attack surfaces as our initial variables:

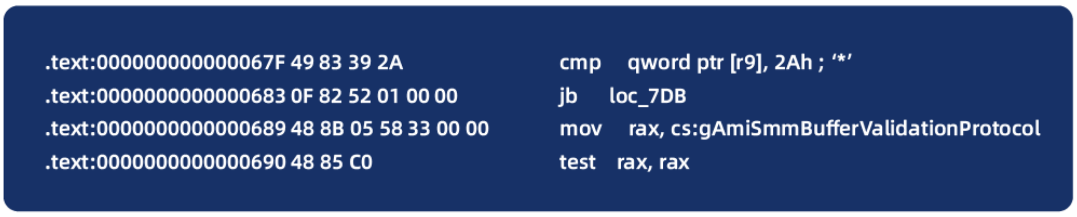
- CommBuffer and CommBufferSize
- ReadSaveState
- Absolute/long jumps below 0xc0000
- gRT and gBS
- Datasize and Data of Getvariable
- Parameters passed from the parent function

6.2.2. Variable Value Prediction Module

To further enhance the precision of our analysis, the tool offers a variable numerical prediction functionality through the analysis of assembly instruction blocks.

The code snippet below exemplifies a typical assembly instruction in comparison operations. The CMP instruction compares the data stored in the memory address corresponding to register r9 against the value 0x2A. Only if the comparison result is greater than or equal to 0x2A, the content at position 0x689 will be executed. Therefore, we can predict that within the instruction segment starting from 0x689 as determined by the conditional statement, the value of the use-def variable corresponding to [r9] must be greater than or equal to 0x2A.

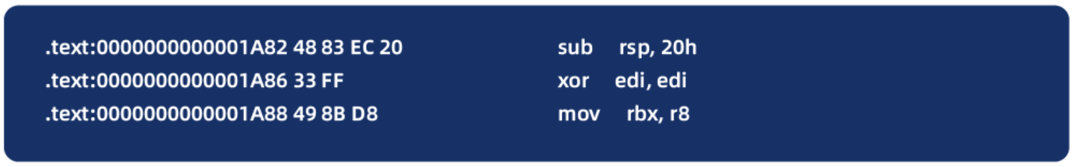
The provided assembly instruction CMP in figure 7 is a typical comparison operation, which compares the data stored in the memory address corresponding to register r9 with 0x2A. Only if it is greater than or equal to 0x2A, the content at the address 0x689 will be executed. Hence, we can infer that within the instruction segment starting from 0x689, the value of the use-def variable r9 must be greater than or equal to 0x2A.



```
.text:000000000000067F 49 83 39 2A              cmp    qword ptr [r9], 2Ah ; '*'
.text:0000000000000683 0F 82 52 01 00 00        jb     loc_7DB
.text:0000000000000689 48 8B 05 58 33 00 00     mov    rax, cs:gAmiSmmBufferValidationProtocol
.text:0000000000000690 48 85 C0                 test   rax, rax
```

Figure 7

Figure 8 shows a typical assembly instruction for XOR (exclusive OR) logical operation. The XOR instruction at the address 0x1A86 performs XOR operation on the EDI register. It can be inferred that regardless of the original value in the EDI register, the value after this logical operation will be equal to 0.



```
.text:0000000000001A82 48 83 EC 20              sub    rsp, 20h
.text:0000000000001A86 33 FF                    xor    edi, edi
.text:0000000000001A88 49 8B D8                 mov    rbx, r8
```

Figure 8

Similar instances of instructions include register self-decrement, operating Logical AND with a constant, multiplying a register by 0, and writing constants into registers, among others.

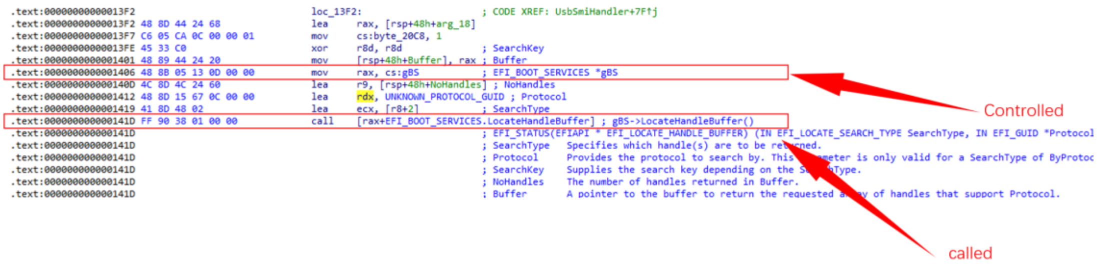
6.3 Detection Plugins

Building upon the data flow tracing analysis engine discussed earlier, we have developed seven plugins, including TOC-TOU and SMMOOB, each tailored to detect one of the seven types of weaknesses presented in UEFI.

6.3.1 Callout Detection Plugin

The callout vulnerability essentially occurs when the destination address of a jump instruction is controllable outside of SMM. To identify such vulnerabilities, we can examine the function pointers at the occurrence of CALL instructions and determine if they are susceptible to external control.

For instance, figure 9 illustrates an n-day vulnerability, wherein an externally controllable data “gBS” is loaded into the “rax” register. Subsequently, during the execution of the call, this vulnerability utilizes the SMM external controllable data in the form of “rax+offset” and incorporates it as part of the destination address.



```
.text:00000000000013F2                              loc_13F2:                               ; CODE XREF: UsbSmiHandler+7F↑j
.text:00000000000013F2 48 8D 44 24 68                lea     rax, [rsp+48h+arg_18]
.text:00000000000013F7 C6 05 CA 0C 00 00 01          mov     cs:byte_20C8, 1
.text:00000000000013FE 45 33 C0                      xor     r8d, r8d        ; SearchKey
.text:0000000000001401 48 89 44 24 20                mov     [rsp+48h+Buffer], rax ; Buffer
.text:0000000000001406 48 8B 05 13 0D 00 00          mov     rax, cs:gBS     ; EFI_BOOT_SERVICES *gBS
.text:000000000000140D 4C 8D 4C 24 60                lea     r9, [rsp+48h+NoHandles] ; NoHandles
.text:0000000000001412 48 8D 15 67 0C 00 00          lea     rdx, UNKNOWN_PROTOCOL_GUID ; Protocol
.text:0000000000001419 41 8D 48 02                   lea     ecx, [r8+2]     ; SearchType
.text:000000000000141D FF 90 38 01 00 00             call    [rax+EFI_BOOT_SERVICES.LocateHandleBuffer] ; gBS->LocateHandleBuffer()
.text:000000000000141D                                                       ; EFI_STATUS(EFIAPI * EFI_LOCATE_HANDLE_BUFFER) (IN EFI_LOCATE_SEARCH_TYPE SearchType, IN EFI_GUID *Protocol
.text:000000000000141D                                                       ; SearchType   Specifies which handle(s) are to be returned.
.text:000000000000141D                                                       ; Protocol     Provides the protocol to search by. This parameter is only valid for a SearchType of ByProtoc
.text:000000000000141D                                                       ; SearchKey    Supplies the search key depending on the SearchType.
.text:000000000000141D                                                       ; NoHandles    The number of handles returned in Buffer.
.text:000000000000141D                                                       ; Buffer       A pointer to the buffer to return the requested array of handles that support Protocol.
```

Figure 9

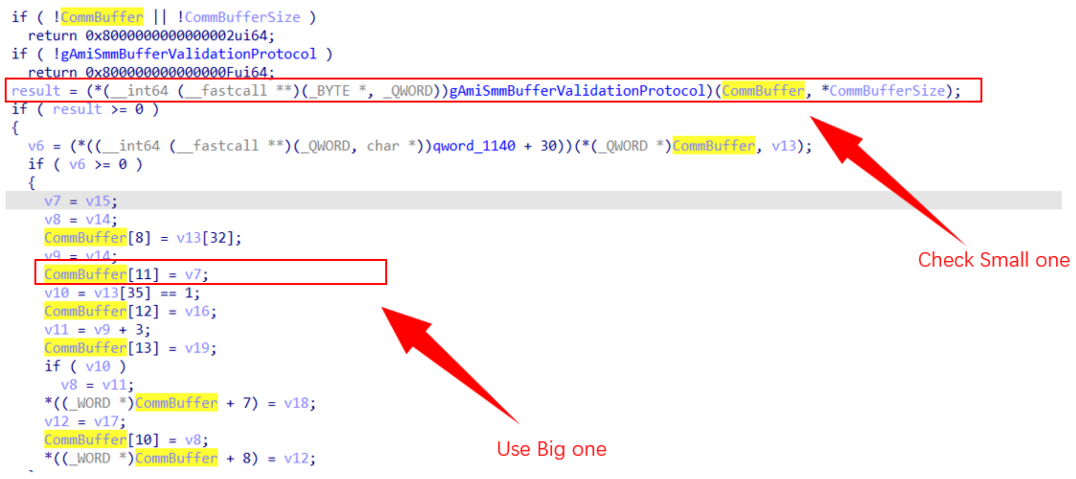
6.3.2. SMM OOB Detection Plugin

SMM Out-of-Bounds (OOB) is a prevalent security concern, and developers often utilize functions such as SmmBufferValidation or gAmiSmmBufferValidation to verify whether the CommBuffer parameter overlaps with SMM to avoid this security issue.

When both the starting address and the size of CommBuffer do not overlap with SMM, the function returns EFI_SUCCESS, signifying successful validation. However, during practice, if the accessed address is CommBuffer+offset, and the value of offset exceeds CommBufferSize, it could lead to the destination address overlapping with SMM, causing security issues.

Our plugin can identify potential SmmBufferValidation functions, acquire the value of CommBufferSize, and within the namespace where CommBuffer is used as a local variable, this plugin will verify whether the offset value of CommBuffer would exceed CommBufferSize to determine the existence of SMM OOB vulnerabilities.

For instance, figure 10 shows a 0-day vulnerability we discovered within the namespace of CommBuffer variables, where the offset values include 8, 11, 12, 13, 2*7, 2*8. This plugin will assess whether the maximum CommBufferSize is greater than all the offset values, otherwise it would result in a vulnerability.

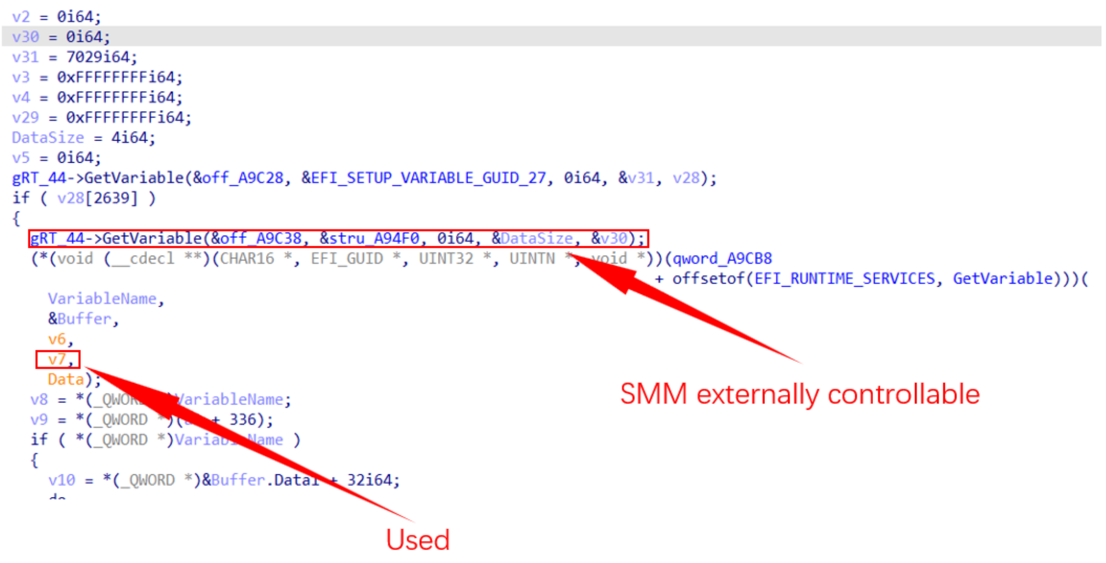


```
if ( !CommBuffer || !CommBufferSize )
  return 0x8000000000000002ui64;
if ( !gAmiSmmBufferValidationProtocol )
  return 0x800000000000000Fui64;
result = (*(__int64 (__fastcall **)(_BYTE *, _QWORD))gAmiSmmBufferValidationProtocol)(CommBuffer, *CommBufferSize);
if ( result >= 0 )
{
  v6 = (*((__int64 (__fastcall **)(_QWORD, char *))qword_1140 + 30))(*(_QWORD *)CommBuffer, v13);
  if ( v6 >= 0 )
  {
    v7 = v15;
    v8 = v14;
    CommBuffer[8] = v13[32];
    v9 = v14;
    CommBuffer[11] = v7;
    v10 = v13[35] == 1;
    CommBuffer[12] = v16;
    v11 = v9 + 3;
    CommBuffer[13] = v19;
    if ( v10 )
      v8 = v11;
    *((_WORD *)CommBuffer + 7) = v18;
    v12 = v17;
    CommBuffer[10] = v8;
    *((_WORD *)CommBuffer + 8) = v12;
```

Figure 10

6.3.3. Overflow Detection Plugins for Getvariable, CopyMem

Our tool can effectively identify buffer overflow vulnerabilities by evaluating whether the parameter DataSize of the Getvariable function is provided by SMM external controllable data. The code snippet shown below illustrates a vulnerability we detected, where the DataSize variable becomes externally controlled at the first highlighted box, and subsequently used at the second box as the parameter of Getvariable. This results in a controlled DataSize, leading to an overflow.



```
v2 = 0i64;
v30 = 0i64;
v31 = 7029i64;
v3 = 0xFFFFFFFFi64;
v4 = 0xFFFFFFFFi64;
v29 = 0xFFFFFFFFi64;
DataSize = 4i64;
v5 = 0i64;
gRT_44->GetVariable(&off_A9C28, &EFI_SETUP_VARIABLE_GUID_27, 0i64, &v31, v28);
if ( v28[2639] )
{
  gRT_44->GetVariable(&off_A9C38, &stru_A94F0, 0i64, &DataSize, &v30);
  (*(void (__cdecl **)(CHAR16 *, EFI_GUID *, UINT32 *, UINTN *, void *))(qword_A9CB8
                                                                        + offsetof(EFI_RUNTIME_SERVICES, GetVariable)))(
    VariableName,
    &Buffer,
    v6,
    v7,
    Data);
  v8 = *(_QWORD *)VariableName;
  v9 = *(_QWORD *)([illegible] + 336);
  if ( *(_QWORD *)VariableName )
  {
    v10 = *(_QWORD *)&Buffer.Data1 + 32i64;
```

Figure 11

Notably, a buffer overflow can also occur when SMM external controllable data is copied. Therefore, we apply the similar method used to detect SMM OOB vulnerabilities in identifying this type of vulnerability. When we observe that the parameter Size of CopyMem originates from externally controllable data, we recognize the existence of such a vulnerability.

6.3.4. TOCTOU (CWE-367) Detection Plugin

Inexperienced developers often assume that implementing conditional checks to validate input data would suffice to prevent vulnerabilities. However, such simplistic checks do not always prove effective. Careless validation can be bypassed through race conditions, as exemplified by code commonly found in antivirus software drivers from a decade ago.

```
UNICODE_STRING filename = Irp->AssociatedIrp.SystemBuffer; // filename.Buffer is pointed to an user controlled address.
if(is_malicious(filename)){
   block();
   alert();
}else{
   pass();
}
```

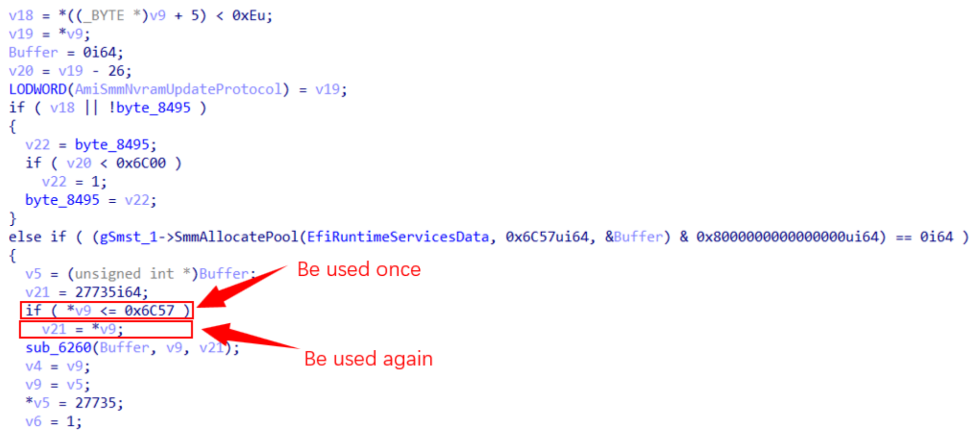
Figure 12

In antivirus software, ring3 hooks typically pass a filename through an IOCTL and notify the kernel to perform certain checks, as shown in figure 12. However, the code lacks an atomic operation for the filename.Buffer variable. By launching another thread before the if (is_malicious(filename)) from the original process is executed and modifying the memory pointed to by filename.Buffer, malicious actors can create a race condition. The attacker can manipulate the value of filename.Buffer to represent a valid file path, thereby evading detection. This manipulation exemplifies a classic instance of TOCTOU operation in cybersecurity.

TOCTOU vulnerabilities also exist in the UEFI. For instance, when SMI is triggered, though the CPU is occupied, various external devices, such as graphics cards, hard drives, and thunderbolt devices, can still access memory through DMA. Exploiting this feature, attackers can construct race conditions during SMI calls. DMA, while incapable of directly accessing SMRAM, can still read and write the addresses of parameters used during SMI calls, thereby bypassing certain security checks in UEFI, constituting a UEFI TOCTOU vulnerability.

In batch testing, we have identified TOCTOU issues in numerous SMI handlers for different modules. Consequently, we have developed a dedicated plugin to detect TOCTOU vulnerabilities in EfiDrill. The detection principle is as followed: when input data outside of SMM is used as a pointer, determine whether the memory corresponding to the same offset relative to this pointer is used multiple times. By employing this method, we can identify a specific type of vulnerability.

As illustrated in the code snippet below, we discovered a 0-day vulnerability. The memory of *v9 is checked in the first red box to be less than 0x6C57, and in the second red box, the memory of *v9 is utilized. However, since v9 originates from Data returned by ReadSaveState, it cannot guarantee that its data remains the same during both usages. This disparity in data when retrieved on two separate occasions (the first time with memory data less than 0x6C57 and the second time with memory data greater than 0x6C57) can lead to critical issues.



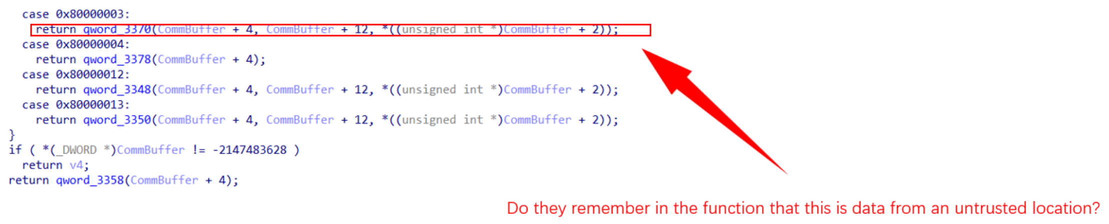
```
v18 = *((_BYTE *)v9 + 5) < 0xEu;
v19 = *v9;
Buffer = 0i64;
v20 = v19 - 26;
LODWORD(AmiSmmNvramUpdateProtocol) = v19;
if ( v18 || !byte_8495 )
{
  v22 = byte_8495;
  if ( v20 < 0x6C00 )
    v22 = 1;
  byte_8495 = v22;
}
else if ( (gSmst_1->SmmAllocatePool(EfiRuntimeServicesData, 0x6C57ui64, &Buffer) & 0x8000000000000000ui64) == 0i64 )
{
  v5 = (unsigned int *)Buffer;
  v21 = 27735i64;
  if ( *v9 <= 0x6C57 )
    v21 = *v9;
  sub_6260(Buffer, v9, v21);
  v4 = v9;
  v9 = v5;
  *v5 = 27735;
  v6 = 1;
```

Figure 13

6.3.5. Weakness Detection Plugin for Identifying External Input in Function Pointer Parameters

In certain portions of the code, there exists a practice of utilizing data from the SMM external environment as function pointer parameters. However, the actual handler of the function pointer may not always be aware of the untrusted nature of this parameter, leading to potential issues. When the function pointer call occurs, our tool examines whether the variables in its parameters are derived from untrusted data outside of SMM to detect such vulnerability risks.

For instance, the code snippet shown below represents a vulnerability we discovered. In this indirect call, data from CommBuffer+4 is passed as argument a1 to the function pointer call, prompting our tool to issue a warning at this point.

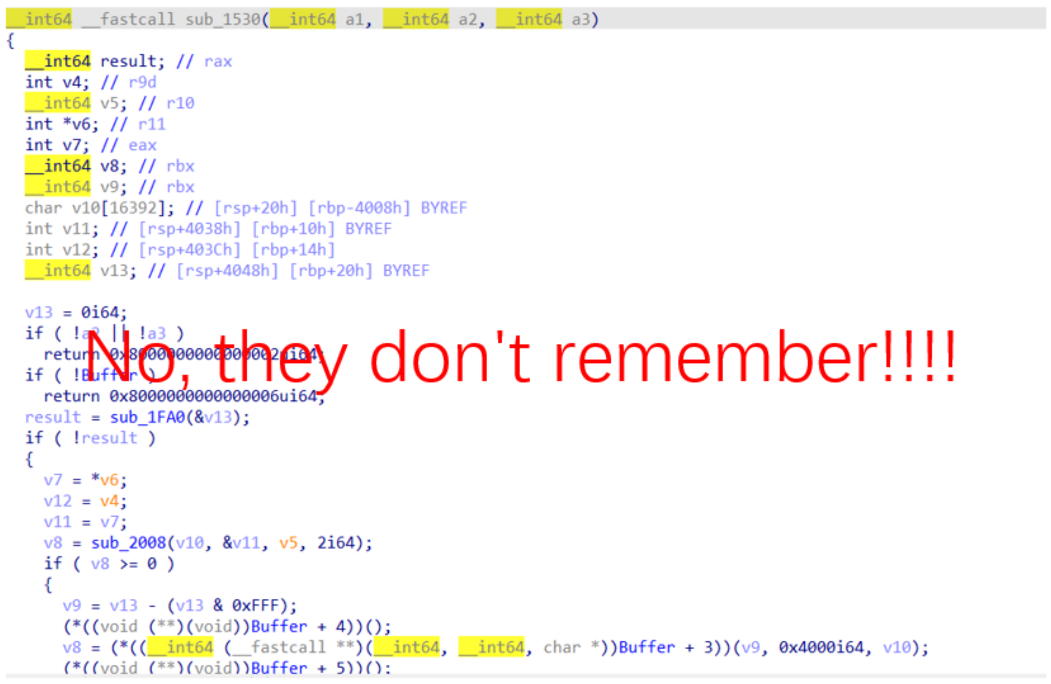


```
  case 0x80000003:
    return qword_3370(CommBuffer + 4, CommBuffer + 12, *((unsigned int *)CommBuffer + 2));
  case 0x80000004:
    return qword_3378(CommBuffer + 4);
  case 0x80000012:
    return qword_3348(CommBuffer + 4, CommBuffer + 12, *((unsigned int *)CommBuffer + 2));
  case 0x80000013:
    return qword_3350(CommBuffer + 4, CommBuffer + 12, *((unsigned int *)CommBuffer + 2));
}
if ( *(_DWORD *)CommBuffer != -2147483628 )
  return v4;
return qword_3358(CommBuffer + 4);
```

Do they remember in the function that this is data from an untrusted location?

Figure 14

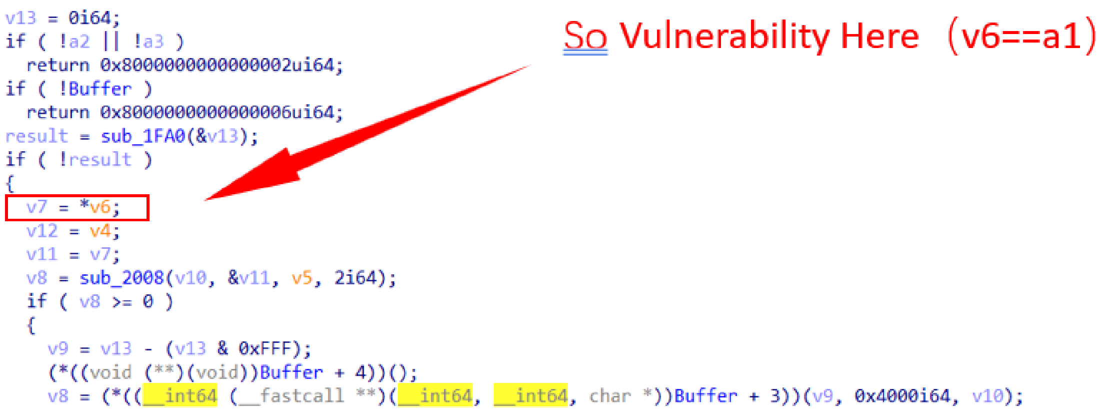
As illustrated in figure 15 and figure 16, through reverse engineering, we ascertain that the function pointer call eventually enters sub_1530, and the developer forgets that this data originates from an untrusted area. The parameter a1 in this function is used as a pointer without undergoing any security checks.



```
__int64 __fastcall sub_1530(__int64 a1, __int64 a2, __int64 a3)
{
  __int64 result; // rax
  int v4; // r9d
  __int64 v5; // r10
  int *v6; // r11
  int v7; // eax
  __int64 v8; // rbx
  __int64 v9; // rbx
  char v10[16392]; // [rsp+20h] [rbp-4008h] BYREF
  int v11; // [rsp+4038h] [rbp+10h] BYREF
  int v12; // [rsp+403Ch] [rbp+14h]
  __int64 v13; // [rsp+4048h] [rbp+20h] BYREF

  v13 = 0i64;
  if ( !a2 || !a3 )
    return 0x8000000000000002ui64;
  if ( !Buffer )
    return 0x8000000000000006ui64;
  result = sub_1FA0(&v13);
  if ( !result )
  {
    v7 = *v6;
    v12 = v4;
    v11 = v7;
    v8 = sub_2008(v10, &v11, v5, 2i64);
    if ( v8 >= 0 )
    {
      v9 = v13 - (v13 & 0xFFF);
      (*((void (**)(void))Buffer + 4))();
      v8 = (*((__int64 (__fastcall **)(__int64, __int64, char *))Buffer + 3))(v9, 0x4000i64, v10);
      (*((void (**)(void))Buffer + 5))();
```

No, they don't remember!!!!

Figure 15



```
v13 = 0i64;
if ( !a2 || !a3 )
  return 0x8000000000000002ui64;
if ( !Buffer )
  return 0x8000000000000006ui64;
result = sub_1FA0(&v13);
if ( !result )
{
  v7 = *v6;
  v12 = v4;
  v11 = v7;
  v8 = sub_2008(v10, &v11, v5, 2i64);
  if ( v8 >= 0 )
  {
    v9 = v13 - (v13 & 0xFFF);
    (*((void (**)(void))Buffer + 4))();
    v8 = (*((__int64 (__fastcall **)(__int64, __int64, char *))Buffer + 3))(v9, 0x4000i64, v10);
```

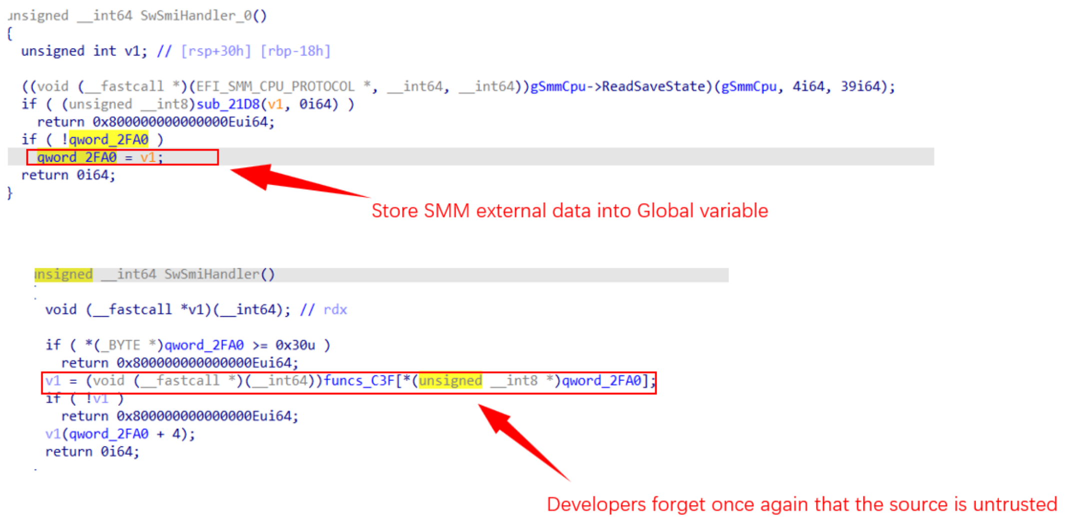
Figure 16

6.3.6 SMM External Untrusted Data Stored in Global Variable Weakness Detection Plugin

In certain sections of the code, there are instances where untrusted data from outside of SMM is assigned to SMM's global variables for future use. However, developers may not always be aware of whether the values within these global variables can be trusted, leading to the absence of security checks on the values of these global variables when they are utilized.

Our tool can determine whether a controllable pointer outside of SMM is passed into a global variable, thereby identifying potential weakness risks.

Figure 17 shows the 0-day vulnerabilities we discovered. When untrusted data is assigned to a global variable and that global variable is subsequently treated as trusted data, it leads to security risks. In the case where data is stored in qword_2FA0, as it originates from Data, a parameter, returned by ReadSaveState, we can infer that it is controllable data from outside of SMM.



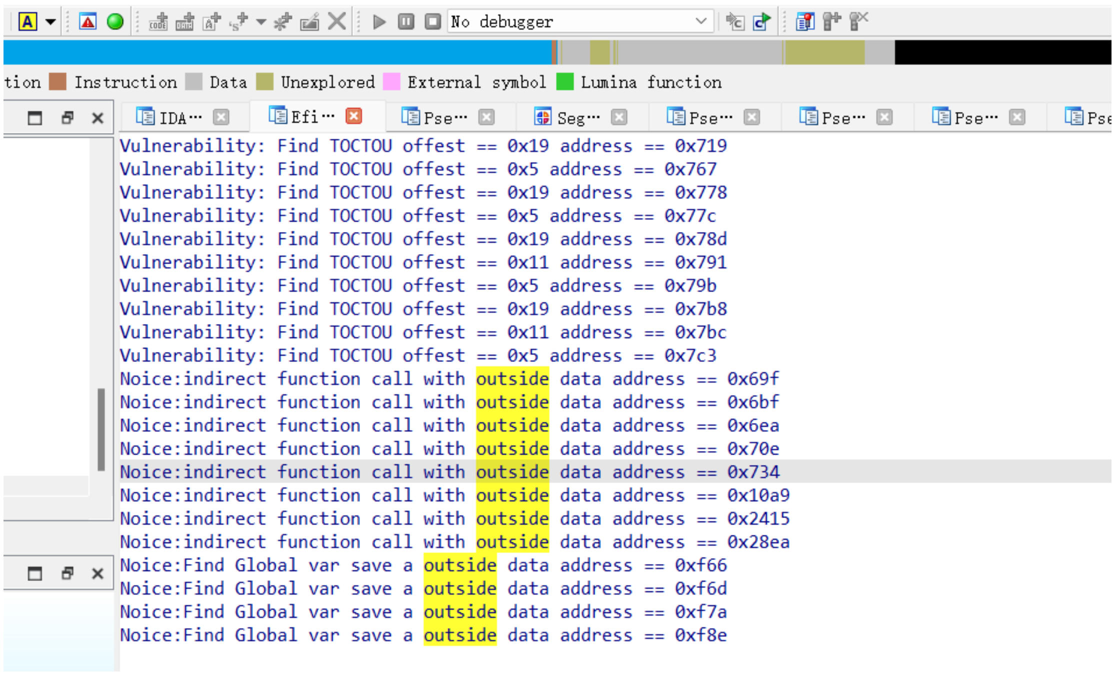
```
unsigned __int64 SwSmiHandler_0()
{
  unsigned int v1; // [rsp+30h] [rbp-18h]

  ((void (__fastcall *)(EFI_SMM_CPU_PROTOCOL *, __int64, __int64))gSmmCpu->ReadSaveState)(gSmmCpu, 4i64, 39i64);
  if ( (unsigned __int8)sub_21D8(v1, 0i64) )
    return 0x800000000000000Eui64;
  if ( !qword_2FA0 )
   qword_2FA0 = v1;
  return 0i64;
}
```

```
unsigned __int64 SwSmiHandler()

  void (__fastcall *v1)(__int64); // rdx

  if ( *(_BYTE *)qword_2FA0 >= 0x30u )
    return 0x800000000000000Eui64;
  v1 = (void (__fastcall *)(__int64))funcs_C3F[*(unsigned __int8 *)qword_2FA0];
  if ( !v1 )
    return 0x800000000000000Eui64;
  v1(qword_2FA0 + 4);
  return 0i64;
```

Figure 17

6.3.7. Vulnerability Detection Report

We offer two distinct methods for generating reports. Users can view a unified-style pop-up window within IDA that presents issues detected by various plugins. Double-clicking on an item in the issue list allows direct navigation to the corresponding code section. Additionally, we support exporting data to report files, facilitating further automated development.



```
Vulnerability: Find TOCTOU offest == 0x19 address == 0x719
Vulnerability: Find TOCTOU offest == 0x5 address == 0x767
Vulnerability: Find TOCTOU offest == 0x19 address == 0x778
Vulnerability: Find TOCTOU offest == 0x5 address == 0x77c
Vulnerability: Find TOCTOU offest == 0x19 address == 0x78d
Vulnerability: Find TOCTOU offest == 0x11 address == 0x791
Vulnerability: Find TOCTOU offest == 0x5 address == 0x79b
Vulnerability: Find TOCTOU offest == 0x19 address == 0x7b8
Vulnerability: Find TOCTOU offest == 0x11 address == 0x7bc
Vulnerability: Find TOCTOU offest == 0x5 address == 0x7c3
Noice:indirect function call with outside data address == 0x69f
Noice:indirect function call with outside data address == 0x6bf
Noice:indirect function call with outside data address == 0x6ea
Noice:indirect function call with outside data address == 0x70e
Noice:indirect function call with outside data address == 0x734
Noice:indirect function call with outside data address == 0x10a9
Noice:indirect function call with outside data address == 0x2415
Noice:indirect function call with outside data address == 0x28ea
Noice:Find Global var save a outside data address == 0xf66
Noice:Find Global var save a outside data address == 0xf6d
Noice:Find Global var save a outside data address == 0xf7a
Noice:Find Global var save a outside data address == 0xf8e
```

Figure 18

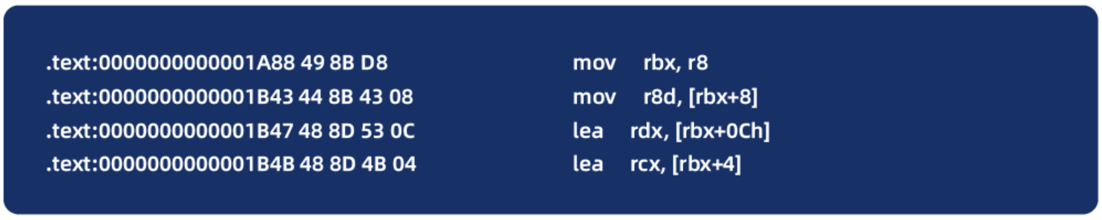
6.4 Automated Analysis Module for Structure Types

During fuzz or reverse engineering, a critical aspect lies in ascertaining the structured\ type of SMM external input data, as this understanding aids in discerning the appropriate utilization of said data and identifying potential issues in the code responsible for its handling. For a structure, its internal variable attributes primarily consist of offsets and types.

First, we conduct an analysis of the internal variable offsets within the structure. For a memory access assembly instruction, it comprises the following attributes:

- Registers
- Offsets

For instance, consider the assembly instruction depicted in figure 19.



```
.text:0000000000001A88 49 8B D8          mov    rbx, r8
.text:0000000000001B43 44 8B 43 08       mov    r8d, [rbx+8]
.text:0000000000001B47 48 8D 53 0C       lea    rdx, [rbx+0Ch]
.text:0000000000001B4B 48 8D 4B 04       lea    rcx, [rbx+4]
```

Figure 19

At address 0x1A88, the instruction signifies the assignment of the value stored in the r8 register to the rbx register. Notably, the r8 register here corresponds to ChilSmiHandler’s CommBuffer, and this instruction reveals that rbx also points to the external memory of the SMM.

The instruction "mov r8d,[rbx+8]" at memory address 0x1B43 signifies the assignment of the value stored in memory at the offset of 8 bytes from the address held in the rbx register to the r8d register. Regarding the memory at the offset of 8 bytes within this structure, we speculate the presence of a variable named varC, and through this assembly instruction, we ascertain that varC is stored into r8d, then we record this information as <use-def var(rbx),8,use-def var(r8d)>.

Applying the same approach to analyze 0x1B4B, we can infer the possible existence of a variable named varB at the structure’s offset of 4 bytes, and the data of this variable is stored in rcx, then we can record this information as <use-def var(rbx),4,use-def var(rcx)>.

Therefore, we predict that there are three internal variables within a structure pointed by rbx, as illustrated in figure 20.

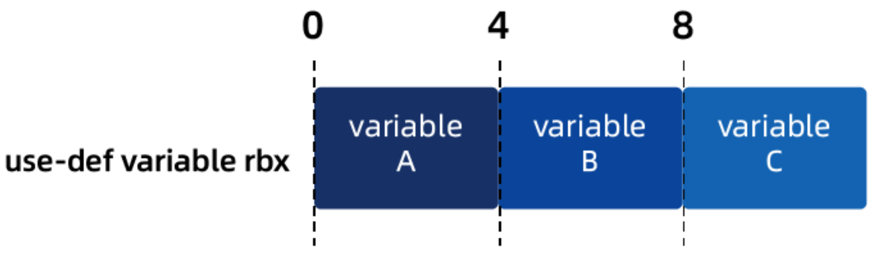


Figure 20

If a variable (e.g., rcx) is used as a pointer again, we can continue the aforementioned analysis by treating the memory pointed to by rcx as a new structure, and repeat this process until all structures have been analyzed. In doing so, we achieve the analysis of internal variable offsets within the structure.

In the subsequent steps, we analyze the internal variable types within the structure, where the uniqueness of assembly instructions classifies the internal variables into two categories: pointer variables and non-pointer variables, with the latter representing a special structure comprising only one internal variable.

As illustrated in the following diagram, the left side assumes a structure containing three variables, but their types remain uncertain. Suppose variable A’s value is extracted to form a new variable D, and variable D is subsequently employed as a pointer in the following assembly instructions; in such a scenario, variable A must be of pointer. Likewise, if variable C's value is extracted to create a new variable E, and the newly derived variable E is not used as a pointer, then variable C's type must be a non-pointer, indicating a regular variable. Through this analytical method, it is possible to discern the types of internal variables within the structure.

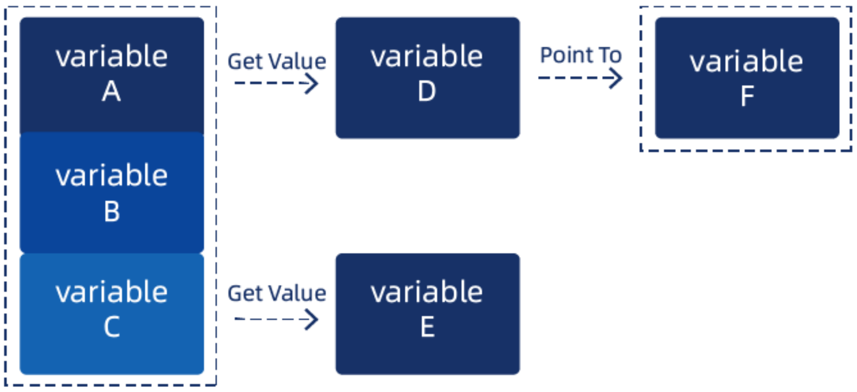


Figure 21

6.5 Extension of EfiDrill

To facilitate future modifications and extensions of our plugin, we have reserved several callbacks and interfaces for extension purposes.

By inheriting from the reserved User_Function_Define interface and implementing the following methods, we can customize the analysis and processing callbacks of EfiDrill:

- function_call_fix: To handle and repair external calls.
- user_def_check: To modify the processing logic of initialized variables.

Moreover, we can add new detection plugins by inheriting from the Base_plugin interface and implementing the following methods to add custom detection rules:

- add_interesting_memory_map_list: To add new tracking variable callbacks.
- vulnerability_find: To execute vulnerability search callbacks.
- copy_use_var: To handle function parameter passing callbacks.
- finish_work: To execute the plugin's lifecycle termination callbacks.

7. Achievements

We have discovered multiple hitherto undisclosed and unreported vulnerabilities, with some even originating from UEFI firmware vendors. Therefore, there is reason to believe that they are widely prevalent across various firmware from different vendors, as indicated in the following list.

- Vulnerability in DELL T7920
- Vulnerability in DELL T7920
- Vulnerability in DELL T7910
- OOB Vulnerability in ASUS D900MD
- OOB Vulnerability in ASUS D900MD

- INTEL-W54FWSHJ
- INTEL-8T9BT8MW
- INTEL-G4O15ERY
- INTEL-T0WK2MUJ
- INTEL-EUCZ5F1V

In addition, we have identified some supply chain issues that might have been previously discovered on other platforms. However, there are still cases where old versions of firmware are in use, leading to known but unpatched vulnerabilities or instances of flawed fixes. These old-version firmware might not have been compiled by the vendors themselves, making it equally challenging to detect them through hash algorithm.

- INTEL-6AYUD87U
- INTEL-VAJCIV59
- CVE-2021-33164（ASUS）

8. Future Plans

Currently, our tool does not support cross-architecture analysis. However, through reserved interfaces, we can enable future support for ARM or LoongArch.
Furthermore, our numerical prediction capability still requires improvement, and by refining it, we can further reduce false positives.

9. References

1. Binarly Advisories
2. Binarly Blog
3. CVE-2022-21198
4. The Memory Sinkhole in BlackHat USA 2015
5. efi_fuzz
6. efiXplorer (Special Note: EfiDrill integrates efiXplorer for obtaining GUIDs.)
7. static program analysis，SPA
8. Reaching definitions in data flow analysis