How To Cook Cisco
<table>
<thead>
<tr>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
</tr>
<tr>
<td>Cisco Exploitation Milestones</td>
</tr>
<tr>
<td>Cisco Diversity</td>
</tr>
<tr>
<td>Target’s Characteristics</td>
</tr>
<tr>
<td>Cisco IOS Mitigations</td>
</tr>
<tr>
<td>Exploitation. Arbitrary Code Execution</td>
</tr>
<tr>
<td>Exploit Debugging</td>
</tr>
<tr>
<td>Gain Control</td>
</tr>
<tr>
<td>Data Execution Prevention Bypass</td>
</tr>
<tr>
<td>Gadgets Catalog</td>
</tr>
<tr>
<td>Write-4 Primitive</td>
</tr>
<tr>
<td>BLRL Gadgets</td>
</tr>
<tr>
<td>Indirect Call Gadgets</td>
</tr>
<tr>
<td>Multitask Gadget</td>
</tr>
<tr>
<td>Multiload Gadget</td>
</tr>
<tr>
<td>Stack Keeper</td>
</tr>
<tr>
<td>Debug Gadget</td>
</tr>
<tr>
<td>How To Disable DEP</td>
</tr>
<tr>
<td>How Does DEP Work On Powerpc?</td>
</tr>
<tr>
<td>Powerpc 405: TLB Entry</td>
</tr>
<tr>
<td>Powerpc 405: DEP Disable Gadget</td>
</tr>
<tr>
<td>Powerpc E500: TLB Entries</td>
</tr>
<tr>
<td>Staged Shellcode</td>
</tr>
<tr>
<td>Shellcode Hunting</td>
</tr>
<tr>
<td>Packet Fragmentation</td>
</tr>
<tr>
<td>Cod Integrity Checking Bypass</td>
</tr>
<tr>
<td>Payload</td>
</tr>
<tr>
<td>TCL-Shellcode</td>
</tr>
<tr>
<td>Stage 1</td>
</tr>
<tr>
<td>Stage 2</td>
</tr>
<tr>
<td>Shellcode Completion</td>
</tr>
<tr>
<td>Arbitrary Code Execution: GeekPWN 2017 Case</td>
</tr>
<tr>
<td>Conclusions</td>
</tr>
<tr>
<td>Contacts</td>
</tr>
</tbody>
</table>
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 and HoF 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, Michael Lynn revealed his brand-new method of bypassing CheckHeeps 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 Exploit for CVE-2016-1287 that became the winner in the Pwnie for Best Server-Side Bug category.
- EXTRABACON that exploits the SNMP protocol vulnerability and is one of the NSA’s tools.

In 2017, there was another exploit developed for the IKEv1 protocol that abuses the same vulnerability - CVE-2016-1287. In addition, CIA Vault 7 Leak occurred. The leaked information included the ROCEM tool description, an exploit for the vulnerability in the implementation of Cisco Cluster Management Protocol (CMP). Later, Cisco announced that the CVE-2017-3881 vulnerability was fixed.

A few months after this, Artem Kondratenko rediscovered the very same vulnerability and made a one-day 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).
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 arbitrary address (NVRAM corruption)
• write-4 (Process Array)

FTP Server Exploit
Andy Davis
Cisco IOS 12.3(18)
CVE-2007-2586
Stack-Based BoF (CWE-121)
Techniques:
• VTY Shellcode
• Signature-based Shellcode

Router Exploitation
Felix ‘FX’ Lindner
Cisco IOS
CVE-2007-0490
Stack-Based BoF (CWE-121)
Techniques:
• ROP (PowerPC)
• disabling caching
• Disassembling Shellcode
• return2caller
• TclLoader

Cisco Shellcode: All in One
George Nosenko
Cisco IOS/XE
Techniques:
• TclShellcode – Concept of an Image Independent Exploit

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

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

TFTP Exploit OSPF Exploit
Felix ‘FX’ Lindner
Cisco IOS 11.1x-11.3.x
CVE-2002-0813
Heap-Based BoF (CWE-122)
Techniques:
• write a positive value at arbitrary address (NVRAM corruption)
• write-4 (Process Array)

Cisco IOS Shellcode And Exploitation Techniques
Michael Lynn
Cisco IOS 11.1x-11.3.x
Heap-Based BoF (CWE-122)
Techniques:
• overwrite (Timer) linked-list
• CheckHeaps bypass
• TTY/TCB Shellcode

Cisco Shellcodes
Gyan Chawdhary, Varun Uppal
Cisco IOS
Techniques:
• bind shell
• reverse shell
• tinyshell

Killing the Myth of Cisco Diversity
Ang Cyi
Cisco IOS
Techniques:
• Interrupt-Hijack Shellcode
• multistage attack

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

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

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).

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**.

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)**. 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):

1. **Gain Control**
   - Stack-based overflow
   - Heap-based overflow

2. **DEP Bypass**
   - Return Orientated 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 arbitrary code:
   - Bind/Reverse shellcode
   - Disassembling shellcode
   - TclShellcode
   - etc

6. **Completion**
   - Return to caller
   - Abuse scheduler function
   - Infinite loop

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 or a bootloader menu (recovery mode).
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 by nccgroup, but probably you would have to slightly modify its code (see Fig. 3).
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).
Instructions for closing the stack frame are as follows:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 01 00 0C</td>
<td>lwz r0, 0xC(r1)</td>
<td></td>
</tr>
<tr>
<td>7C 08 03 A6</td>
<td>mtlr r0</td>
<td></td>
</tr>
<tr>
<td>38 21 00 08</td>
<td>addi r1, r1, 8</td>
<td></td>
</tr>
<tr>
<td>4E 80 00 20</td>
<td>blr</td>
<td></td>
</tr>
</tbody>
</table>

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)*, 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, bctr.

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

```
_mr r3, r9           # Move Register
lwz r0, 0x14(r1)     # Move to link register
mtlr r0             # Move to link register
lwz r30, 8(r1)       # Move to link register
lwz r31, 0xC(r1)     # Move to link register
addi r1, r1, 0x10    # Move to link register
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)
- 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 (see the listing on the following page).
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 proved to serve the task well, however it misses some types of devices. That is why the 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 Primitive

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

2. mtctr
   bctr
   # Move to count register

3. mtctr
   mr
   bctr
   lwz
   mtlr
   lwz r0, 0x10+arg_4(r1)
   mtlr r0
   # Move to count register
   # Move Register
   # Branch unconditionally
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) # 3rd task
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 privelleges 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).

<table>
<thead>
<tr>
<th>PID (Process ID)</th>
<th>0</th>
<th>23</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>TLBHI (Tag entry)</th>
<th>0</th>
<th>21</th>
<th>22</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EPN</td>
<td>size</td>
<td>V</td>
<td>E</td>
<td>UO</td>
<td>TD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TLBLO (Data entry)</th>
<th>0</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RPN</td>
<td>EX</td>
<td>WR</td>
<td>ZSEL</td>
<td>W</td>
<td>I</td>
<td>M</td>
<td>G</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5. The TLB register structure 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, 1 bit): 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).
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.

```assembly
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:

```assembly
mtctr   r29   # points to write-4 gadget
bctrl   # write 0xffffffff to gZprValue
mtlr   r28   # points to dep-disable gadget
blr    # 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:

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
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**

<table>
<thead>
<tr>
<th>MAS0</th>
<th>TLBSEL, ESEL, NV</th>
<th>Select entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAS1</td>
<td>V, IPROT, TID, TS, TSIZE</td>
<td></td>
</tr>
<tr>
<td>MAS2</td>
<td>EPN (0-31), EPN (32-51), XO, X1, WIMGE</td>
<td></td>
</tr>
<tr>
<td>MAS3</td>
<td>RPN (32-51), UO-3, UX, SX, UW, SW, UR, SR</td>
<td></td>
</tr>
<tr>
<td>MAS4</td>
<td>TLBSELD, TIDSELD, TSIZED, default XOx1, DWIMGE</td>
<td></td>
</tr>
<tr>
<td>MAS5</td>
<td>SPID2, SPID3</td>
<td></td>
</tr>
<tr>
<td>MAS6</td>
<td>SPID0, SPID1, SAS</td>
<td></td>
</tr>
</tbody>
</table>

On chip TLBs

tibwe

Defaults

For Searching
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 caching 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:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Permissions</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x02C00000</td>
<td>0x02FFFFFF</td>
<td>4194304</td>
<td>lomem</td>
</tr>
</tbody>
</table>
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 of the **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.

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit</td>
<td>Header Length</td>
<td>Type of service</td>
<td>Total Length</td>
</tr>
<tr>
<td>Identification</td>
<td>Flags</td>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td>Time to live</td>
<td>Protocol</td>
<td>Checksum</td>
<td></td>
</tr>
<tr>
<td>Source Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options and Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 research it was proposed to reuse the ROMMON code you already have to completely disable caching. 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:

```c
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).
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.
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**
- **Interrupt-Hijack Shellcode**
- **Overwriting Exception Vector**
- **SYNFull Knock**
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).

![Fig. 12.Checksum algorithm](image-url)
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:

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Size(b)</th>
<th>Class</th>
<th>Media</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0x03FFFFFF</td>
<td>67108864</td>
<td>Local</td>
<td>R/W</td>
<td>main</td>
</tr>
<tr>
<td>0x00000020</td>
<td>0x03FFFFFF</td>
<td>67108832</td>
<td>Local</td>
<td>R/W</td>
<td>main:coredump</td>
</tr>
<tr>
<td>0x00003000</td>
<td>0x01715233</td>
<td>24191540</td>
<td>IText</td>
<td>R/W</td>
<td>coredump:text</td>
</tr>
<tr>
<td>0x01800000</td>
<td>0x018FFFFF</td>
<td>1048576</td>
<td>IText</td>
<td>R/W</td>
<td>coredump:dltext</td>
</tr>
</tbody>
</table>

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.
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**
  Invariant is a structure of code.
- **Disassembling Shellcode by Felix ‘FX’ Lindner — Cisco IOS Router Exploitation, 2009**
  Invariant is an unique string.
- **Interrupt-Hijack Shellcode by Columbia University NY — Killing the Myth of Cisco IOS Diversity, 2011.**
  Invariant is an interrupt handler routines.
- **Tcl-Shellcode by George Nosenko — Cisco IOS Shellcode: All-In-One, 2015.**
  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](#).

**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)
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).
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*.

```c
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).

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.
Contacts

Telephone: +1 5103232636
Email: info@embedi.com

Address: 2001 Addison Street
Berkeley, California 94704

www.facebook.com/Embedi
twitter.com/_embedi_

linkedin.com/company/embedi
website: embedi.com