Wrocław University of Science and Technology

Faculty of Information and Communication Technology

List 3 Report

Hakan Yildizhan 270056
Gabriel Wechta 250111

Supervisor:
Wojciech Wodo, PhD

December 4, 2022

Report
Embedded Security

Contents

0 | Introduction

The purpose of the exercises enumerated in the task formulation is learning how to approach embedded device with a goal to get as much access and power as possible. We will use many means and technologies to gather information about our device and perform some unintended actions that potentially could be used in a profitable manner. We will also perform two successful attacks on the device.

As the target device, we have chosen `NETGEAR Wireless-N 150 ADSL2+ Modem Router` (FCC ID PY309300114) [1]. We have chosen this router because it is fairly cheap, has quite detailed documentation, has nicely exposed UART interface, and quite lively reverse engineering community.

1 | List 3

1.1 Task formulation

"You will pick one of the chosen device with embedded system implemented (for instance router, camera, electronic baby sitter, etc.) and try to reverse engineer it. You have to perform ad least:

1. Information and data gathering,
2. UART detection, determine voltage levels and pins order, establish the connection,
3. Get firmware from the board or download it from manufacturer,
4. Analyze firmware, extract it, get access to the essential resources of the device,
5. Retrieve root password or get access to the root shell on the device.
6. Have a look at running OS system on device (after accessing the root) and perform some analysis of working processes, find out how this type of embedded OS is built, what commands are available, access the router config file, try to get some files from the device.

Your task is to overcome each security means by using different analysis methods and prepare scripts/toolkits facilitating attacks. You should be able to establish connection between target board and your computer via UART or other interface in order to perform the communication and take control over the device."

1.2 Information and data gathering

Before acquiring the Netgear DGN1000 router we gathered information about the device, it's components and hacking capabilities. Quick googling returned the device's FCC ID (**PY309300114**). From this point, finding information about device was easy. Websites `https://deviwiki.com` and `http://en.techinfodepot.shoutwiki.com` provided many useful information. Mainly confirmed UART interface presence, and helped to identify onboard chips.

From this websites we also learned default login values, such as:

- IP address: **192.168.0.1**
- login user: **admin**
- login password: **password**

Quick google search also showed four posts about vulnerabilities nicely indexed on `https://www.exploit-db.com` website. We will get back to them later in section 1.10.

1.3 Firmware analysis

We did not have to dump the firmware from the device, luckily we were able to download firmware directly from the Netgear website `https://www.netgear.com/support/product/DGN1000.aspx#download`. Producer confidently offers many versions. We downloaded the newest one. The download zip archive contained two files release file (html) and binary data file.

Next part caused some problems on Ubuntu 20.04. We switched to Kali Linux 5.14.0. Kali has all requested tools, mainly `binwalk` installed out of the box.

We started by examining img file. Examination showed U-Boot version, two LZMA

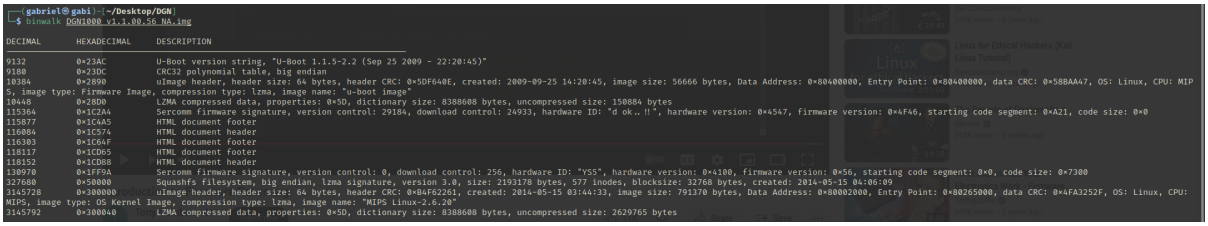


Figure 1.1: Content of the `binwalk` examination.

archives, and one squashed file system. Further by hand examination of LZMA archives finished unfortunately with an uninteresting result – empty `root` directory and `/dev/console` output interface device.

A much nicer way to conduct an examination is doing recursive extraction simply using listing 1.1.

```
binwalk -eM DGN1000_v1.1.00.56_NA.img
```

Listing 1.1: Executing `binwalk` command.

This command attempts to uncompress and unsquash files automatically. For unsquashing it utilizes `sasquatch` command that had to be installed from private repo `https://github.com/devttys0/sasquatch`. After that we extracted typical Linux file system. See picture 1.2.

```
┌──(gabriel㉿gabi)-[~/Desktop/DGN/_DGN1000_v1.1.00.56_NA.img.extracted/squashfs-root]
└─$ ls -l
total 372
lrwxrwxrwx 1 gabriel gabriel      9 Nov 14 20:58 bin → usr/sbin/
lrwxrwxrwx 1 gabriel gabriel      9 Nov 14 20:58 dev → /dev/null
lrwxrwxrwx 1 gabriel gabriel      9 Nov 14 20:58 etc → /dev/null
lrwxrwxrwx 1 gabriel gabriel      9 Nov 14 20:58 home → /dev/null
drwxr-xr-x 2 gabriel gabriel   4096 Apr 20  2009 k2img
drwxr-xr-x 3 gabriel gabriel   4096 May 15  2014 lib
-rwxr-xr-x 1 gabriel gabriel 340744 May 15  2014 modemhwe.bin
drwxr-xr-x 2 gabriel gabriel   4096 Nov 13  2000 proc
lrwxrwxrwx 1 gabriel gabriel      9 Nov 14 20:58 sbin → usr/sbin/
lrwxrwxrwx 1 gabriel gabriel      9 Nov 14 20:58 sys → /dev/null
drwxr-xr-x 4 gabriel gabriel   4096 May 14  2009 tmp
drwxr-xr-x 6 gabriel gabriel   4096 Aug  1  2008 usr
lrwxrwxrwx 1 gabriel gabriel      9 Nov 14 20:58 var → /dev/null
drwxr-xr-x 2 gabriel gabriel   4096 Apr 20  2009 wlan
lrwxrwxrwx 1 gabriel gabriel      9 Nov 14 20:58 www → /dev/null
drwxr-xr-x 2 gabriel gabriel  12288 Nov 14 20:58 www.eng
```

Figure 1.2: Unsquashed file system from downloaded firmware.

As we can see plenty of symlinks pointers were pointing outside of the extraction directory. Therefore, for security reasons, they were automatically changed to point to `/dev/null`.

1.4 Getting inside

The case of this router was held down by 4 screws from the bottom of the case only. We have used special screwdriver to open the case. The inside components of Netgear router are well labeled and easy to read.

Major components are described below.

- CPU – Infineon PSB 50601
- Flash Chip – Macronix MX25L3205DM2I-12G – 4MiB
- RAM Chip – EtronTech EM639165TS-6G – 16MiB
- Switch – Atheros AR8216
- ETH chip – Infineon PSB 50601
- 4 ETH ports - 100MbE-LAN

1.5 Accessing UART

We located UART debugging interface J1 on board, unfortunately pins were not labeled. With the help of a forum post at OpenWRT website here we were able to recognize individual UART pins.

UART localization with pins description can be seen in the picture 1.3. A quick measurement of the VCC pin with a multimeter showed that it is a 3.3V interface.

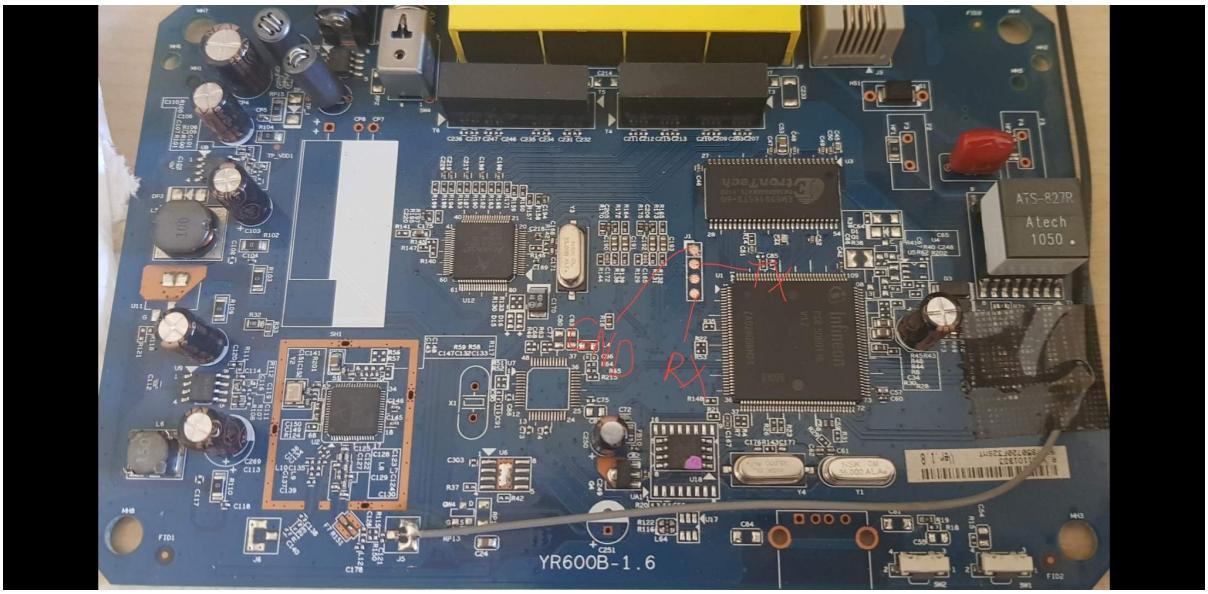


Figure 1.3: UART pins label.

Mr. Błaśkiewicz during laboratory classes used his steady hand and soldered on some pin headers for our convenience, what can be see in the picture 1.4.

Figure 1.4: Doctor Błaśkiewicz's art.

1.6 Getting shell

Luckily our USB to serial TTL has option to specify whether interface is 3.3V or 5V. For connection details see picture 1.5.

After connecting USB, we checked available USB interfaces using `lsusb` command and found that there is a new open interface. Further examination of `/dev` directory showed that there is a no `/dev/ttyUSB0` device. Actually it turned out that this extremely unlucky newest Ubuntu bug.

Ubuntu 22.04 has configuration file located at `/usr/lib/udev/rules.d/85-brltty.rules`, that is most likely default configuration for some braille reader device. In this configuration file there is a record reserving USB bus address for that device, see listing 1.2.

```
ENV{PRODUCT}=="1a86/7523/*", ENV{BRLTTY_BRAILLE_DRIVER}
    ="bm", GOTO="brltty_usb_run"}
```

Listing 1.2: Faulty config line.

`1a86/7523` is also address of the UART converter we have. Commenting the line and rebooting helped.

Although there are many clever ways to find out baud rate we didn't have to use them, we luckily guessed it in the first try – 115200.

We connected to the running device using 1.3.

```
screen -L /dev/ttyUSB0 115200
```

Listing 1.3: Executing `screen` command.

Initially we got empty console, but restarting device showed that it boots correctly and outputs nice logs. At the end of the boot sequence, device asks to press enter in order to get access to shell. That was the reason why on the first try we got empty terminal

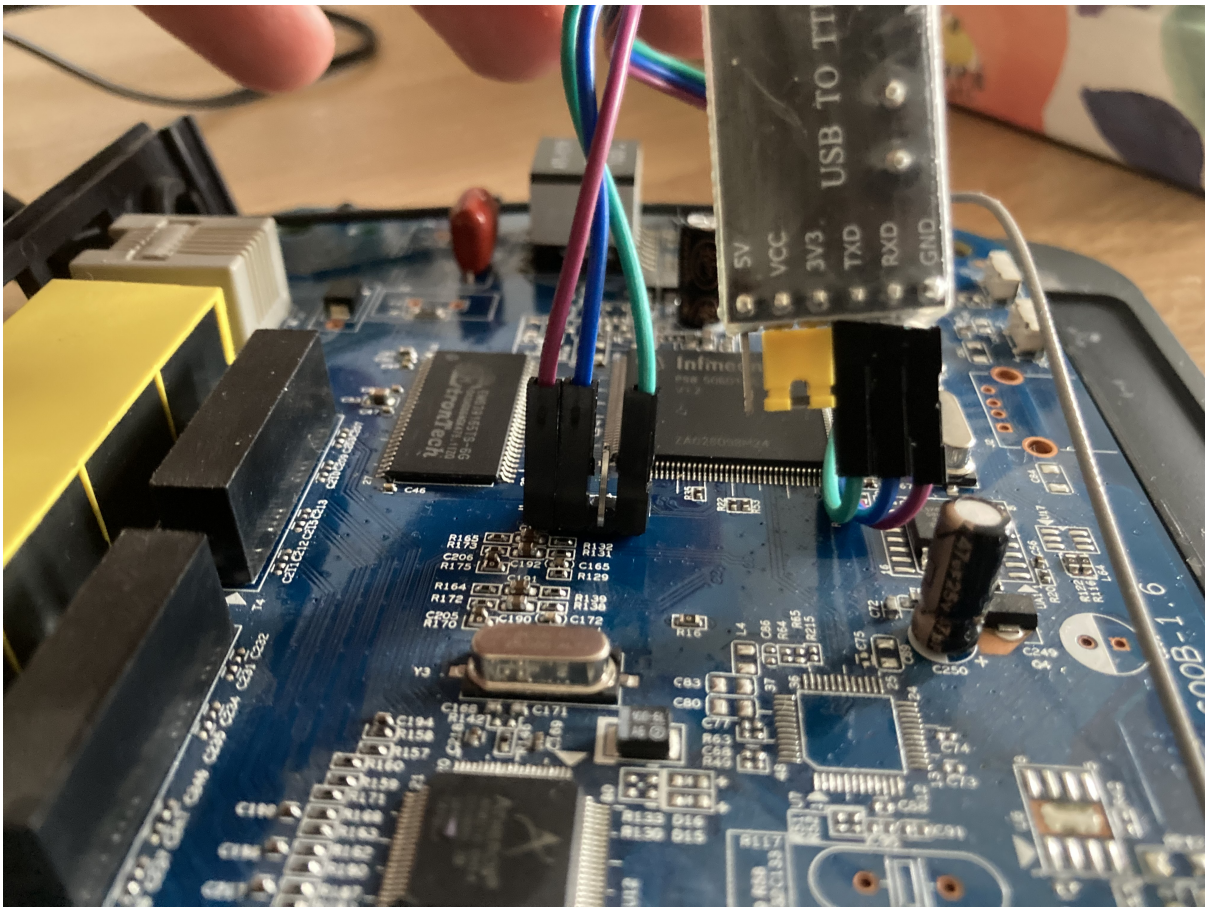


Figure 1.5: UART to serial TTL connection.

with no shell prompt. Additional `-L` option dumped content of the boot logs to the file, so that we could analyze it afterwards.

1.7 Looking around on the running device

We got root shell for free.

First thing we did was examining `etc/passwd`, see picture 1.6. We have examined BusyBox commands, then we took a look at `/usr/sbin` to find out which commands are available. After that we took a glance at `/proc` directory. Very quick examination showed that device supports very broken WPS protocol (see picture 1.7).

We found out plenty of very interesting files to examine and potentially attack, but due to small amount of time on our hands we focused on known vulnerabilities (see 1.10).

1.7.1 Copying file

After very and fearless fight we gave up on trying to copy binary files from the router. But in order to honor our enemy we will describe our journey.

BusyBox on DGN1000 does not have any tool for representing/encoding binary values as readable characters, therefore we could not transmit binary file/bytes as characters. The only tools we found on the device are `cat` which fails to print not-printable characters and `read` what potentially could be used. We had an idea to use `read` to read byte by

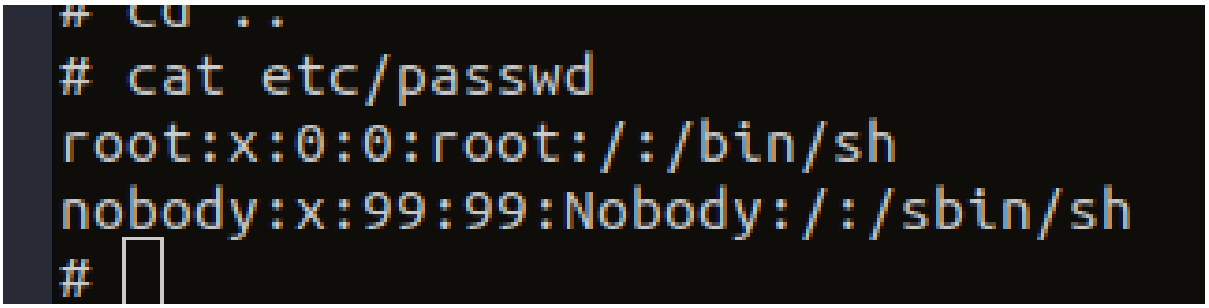


Figure 1.6: `etc/passwd` file.

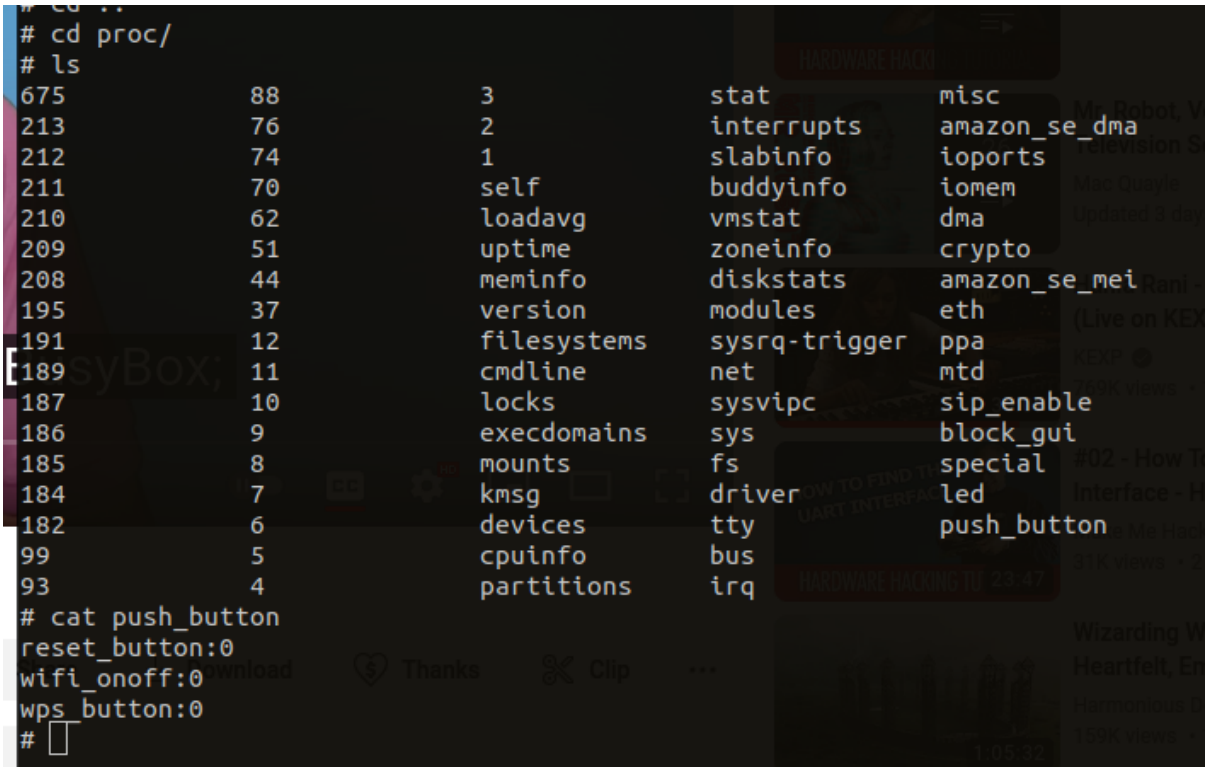
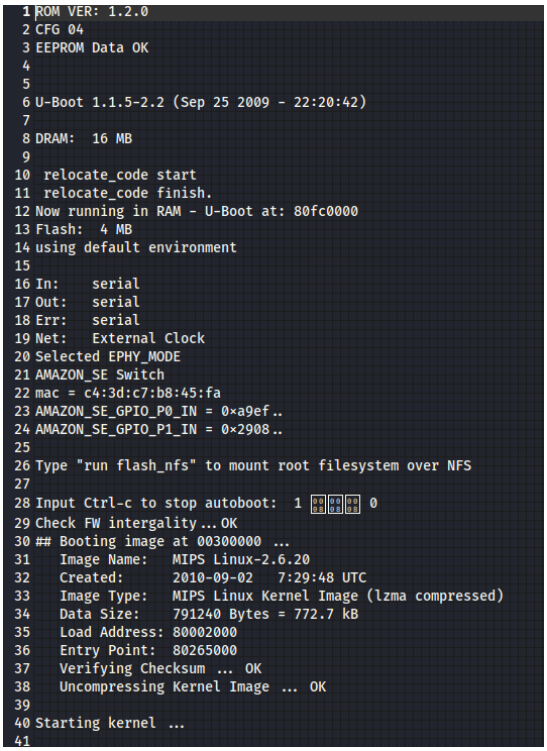


Figure 1.7: `proc` file.

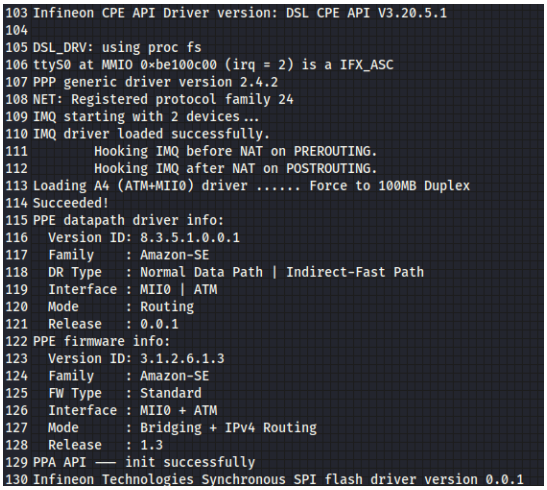
byte from the file and encode it using math operation, simplest bash commands and use `printf` to print it.

We found a script that potentially could do that. Also to achieve, because UART terminal emulation has fixed number of character that it is able to keep in the buffer, which was shorter than the script that we wanted to execute, so we had to save the script on the device. When we tried to perform some write operation it failed with information from the OS that this file system is read-only (due to the fact that squashFS is read-only FS). But we found a solution (great answer: `https://unix.stackexchange.com/questions/313124/make-readonly-etc-writeable`), we union-mounted special directory at `/tmp/etc/upper` what gave us writing possibility.

We created couple of files by echoing some strings into them, because our BusyBox does not have `touch` command. Then we removed all forbidden chars in decoding script and replaced new lines with "\n" and copied it onto the device via echoing to the file. We were so close(!) but unfortunately it turned out that this particular BusyBox `read` command is super limited and does not support any relative flags (namely: -n for number



```
ROM VER: 1.2.0
CFG 04
EEPROM Data OK


U-Boot 1.1.5-2.2 (Sep 25 2009 - 22:20:42)

DRAM:  16 MB

 relocate_code start
 relocate_code finish.
Now running in RAM - U-Boot at: 80fc0000
Flash:  4 MB
using default environment

In:    serial
Out:   serial
Err:   serial
Net:   External Clock
Selected EPHY_MODE
AMAZON_SE Switch
mac = c4:3d:c7:b8:45:fa
AMAZON_SE_GPIO_P0_IN = 0xa9ef..
AMAZON_SE_GPIO_P1_IN = 0x2908..

Type "run flash_nfs" to mount root filesystem over NFS

Input Ctrl-c to stop autoboot:  1 [redacted] 0
Check FW intergality...OK
## Booting image at 00300000 ...
   Image Name:   MIPS Linux-2.6.20
   Created:      2010-09-02   7:29:48 UTC
   Image Type:   MIPS Linux Kernel Image (lzma compressed)
   Data Size:    791240 Bytes = 772.7 kB
   Load Address: 80002000
   Entry Point:  80265000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK

Starting kernel ...

```



```
Infineon CPE API Driver version: DSL CPE API V3.20.5.1

DSL_DRV: using proc fs
ttyS0 at MMIO 0xbe100c00 (irq = 2) is a IFX_ASC
PPP generic driver version 2.4.2
NET: Registered protocol family 24
IMQ starting with 2 devices...
IMQ driver loaded successfully.
        Hooking IMQ before NAT on PREROUTING.
        Hooking IMQ after NAT on POSTROUTING.
Loading A4 (ATM+MII0) driver ..... Force to 100MB Duplex
Succeeded!
PPE datapath driver info:
  Version ID: 8.3.5.1.0.0.1
  Family    : Amazon-SE
  DR Type   : Normal Data Path | Indirect-Fast Path
  Interface : MII0 | ATM
  Mode      : Routing
  Release   : 0.0.1
PPE firmware info:
  Version ID: 3.1.2.6.1.3
  Family    : Amazon-SE
  FW Type   : Standard
  Interface : MII0 + ATM
  Mode      : Bridging + IPv4 Routing
  Release   : 1.3
PPA API -- init successfully
Infineon Technologies Synchronous SPI flash driver version 0.0.1
```

Figure 1.8: The two most interesting boot logs fragments.

of character to written or -d for setting delimiter), therefore script could not be run.
Finally we gave up on trying, but surely there is a way.

1.8 Boot logs examination

We provide some pictures of the boot logs, see pictures 1.8.
From the boot logs we learned that:

- the device has EEPROM chip.
- the boot loader is U-Boot 1.1.5-2.2
- 16 MB RAM
- 4 MB flash memory
- we got device's MAC – `c4:3d:c7:b8:45:fa`
- there is some information about general-purpose input/output pins, probably present on the board
  - `AMAZON_SE_GPIO_P0_IN = 0xa9ef..`
  - `AMAZON_SE_GPIO_P1_IN = 0x2908..`
- there is possibility to stop autoboot, what we will examine later
- the firmware is Linux-based and is running a very old Linux kernel – `Linux version 2.6.20-Amazon_SE (rootBuildServer) (gcc version 3.4.4 20050119 (MIPS SDE)) #31 Thu Sep 2 15:29:44 CST 2010`, also with information where in the memory image is being booted (address: 00300000)

- the line `This model supports only basic BSP feature on Amazon SE Small board with SPI flash USB host and PPE A4 firmware` is not so optimistic and says that this device has only basic functionalities of board support package (BSP)
- information that user-defined physical RAM map is 01000000 to 00000000
- information about memory `13528k/16384k available (2064k kernel code, 2856k reserved, 377k data, 128k init, 0k highmem)`
- number of memory pages – 4064
- information that the device most probably is using Direct Memory Access (DMA)
- squashfs version `3.2-r2 (20070115)`
- very important partition addresses list:
  - `0x00000000-0x00020000 : "U-Boot"`
  - `0x00020000-0x00030000 : "ENV_MAC"`
  - `0x00030000-0x00040000 : "DPF"`
  - `0x00040000-0x00050000 : "NVRAM"`
  - `0x00050000-0x00300000 : "RootFS_DPUMP"`
  - `0x00300000-0x00400000 : "Kernel"`
  - `0x00050000-0x00400000 : "ROOTFS_KERNEL"`
  - `0x00000000-0x00020000 : "POT"`

  What's interesting there is no doubled partition, so most probably device is not able to perform online update, without a need to shut down
- iptables info
- funny line: `All bugs added by David S. Miller <davem@redhat.com>`
- BusyBox version – v1.13.1
- two intrresting error lines:
  - `cat: can't open '/tmp/etc/block.htm': No such file or directory`
  - `cat: can't open '/proc/nvram/BaseMacAddr': No such file or directory`

1.9 Autoboot cancellation

During boot sequence there is a small time gap during which, if one presses `ctr+c` then the autoboot stops. We see `AMAZON_SE` U-boot prompt (picture 1.9). Examination of this command showed that they are quite powerful. We didn't examine it deeply, but perhaps there is a possibility to run the device in factory mode, by using `setenv` command to set some environment variable (something like `factory`, `factory_mod`, we have seen something similar couple of times) to some specific value. Also there are big chances that we could provide malicious firmware image and use one of the update commands to install it on the router.

```
Type "run flash_nfs" to mount root filesystem over NFS

Input Ctrl-c to stop autoboot:  0
AMAZON_SE # <INTERRUPT>
AMAZON_SE # ls
Unknown command 'ls' - try 'help'
AMAZON_SE # help
?       - alias for 'help'
askenv  - get environment variables from stdin
admos- assign MAC address via Upgrade Utility
autoscr - run script from memory
base    - print or set address offset
bdinfo  - print Board Info structure
bootm   - boot application image from memory
bootp   - boot image via network using BootP/TFTP protocol
cmp     - memory compare
cp      - memory copy
crc32   - checksum calculation
dhrystone       - benchmark
download- download image via Upgrade Utility
echo    - echo args to console
eeprom  - EEPROM sub-system
go      - start application at address 'addr'
help    - print online help
httpdownload - httpdownload
loadb   - load binary file over serial line (kermit mode)
loady   - load binary file over serial line (ymodem mode)
loop    - infinite loop on address range
md      - memory display
mm      - memory modify (auto-incrementing)
mtest   - simple RAM test
mw      - memory write (fill)
nm      - memory modify (constant address)
printenv- print environment variables
rarpboot- boot image via network using RARP/TFTP protocol
reset   - Perform RESET of the CPU
run     - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv  - set environment variables
sflash  - SPI FLASH sub-system
sleep   - delay execution for some time
tftpboot- boot image via network using TFTP protocol
upgrade - forward/backward copy memory to pre-defined flash location
version - print monitor version
AMAZON_SE #
```

Figure 1.9: Available commands after canceling autoboot.

1.10 Exploit-DB Vulnerabilities

Website `https://www.exploit-db.com` currently shows four exploits that leverage DGN1000 vulnerabilities. Mainly `setup.cgi` file that allows to perform arbitrarily commands on firmware up to `1.1.00.48` version. The firmware version of our device is `1.1.00.35`, the newest one available on Netgear website is `1.1.00.56`.

We were able to execute following attacks.

1.10.1 Shell command injection

By typing such address `http://192.168.0.1/setup.cgi?next_file=netgear.cfg&todo=syscmd&cmd=cat+/www/.htpasswd&curpath=/&currentsetting.htm=1` into URL bar in the web-browser on a device connected to the router we can execute arbitrary shell commands as root. In this case it is `cat /www/.htpasswd`. See picture 1.10.

The vulnerability discoverer claims that address needs to have `currentsetting.htm=1` substring in order to omit validation, but on our device even that is unnecessary.

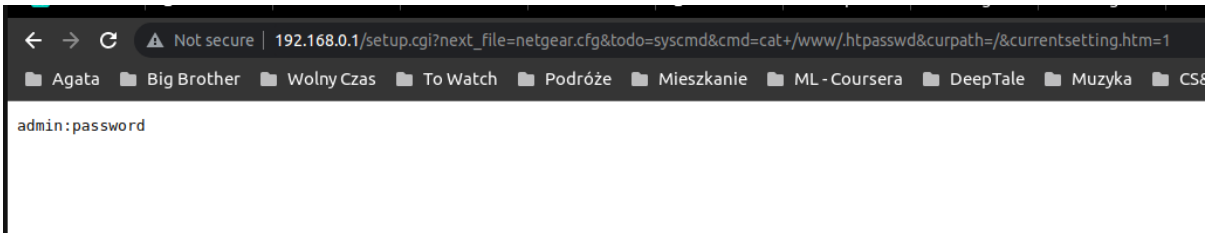


Figure 1.10: Executing `cat /www/.htpasswd` command.

1.10.2 Cross Site Scripting

Many fields in `setup.cgi` request are not properly validate, therefore it allows to insert malicious JavaScript code.

For example the parameter `service_name` can be used to execute `alert(2);` js script. See picture 1.11.

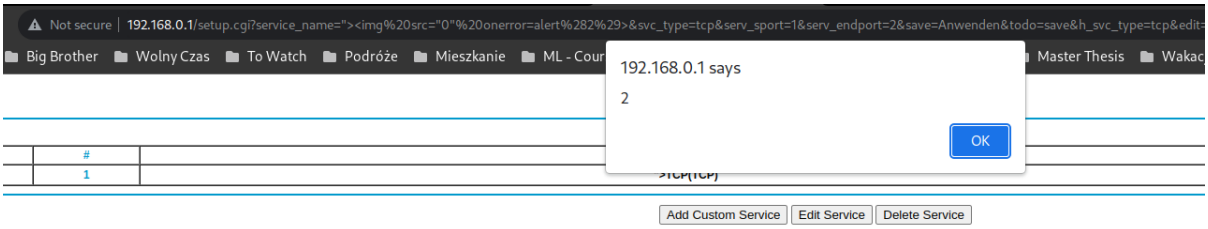


Figure 1.11: Performing XSS using bad validation.

Very nice post, with the vulnerability fix timeline, from the person who discovered this attack and reported it to the vendor can be see here `https://www.exploit-db.com/exploits/24464`.

1.11 Decompilation and Code Analysis via Ghidra

Ghidra is a reverse engineering framework that includes tools to analyze compiled code on a variety of platforms including disassembly, assembly, decompilation, graphing, and scripting.

1.11.1 Preparing for Analysis

Using the firmware file `"DGN1000_V1.1.00.50WW.img"` NetGear kindly serves publicly at URL `https://www.netgear.com/support/product/DGN1000.aspx`, we extracted the file system and identified `/usr/sbin/setup.cgi` file as our target. First step is creating a project and importing aforementioned file, see figure 1.12.

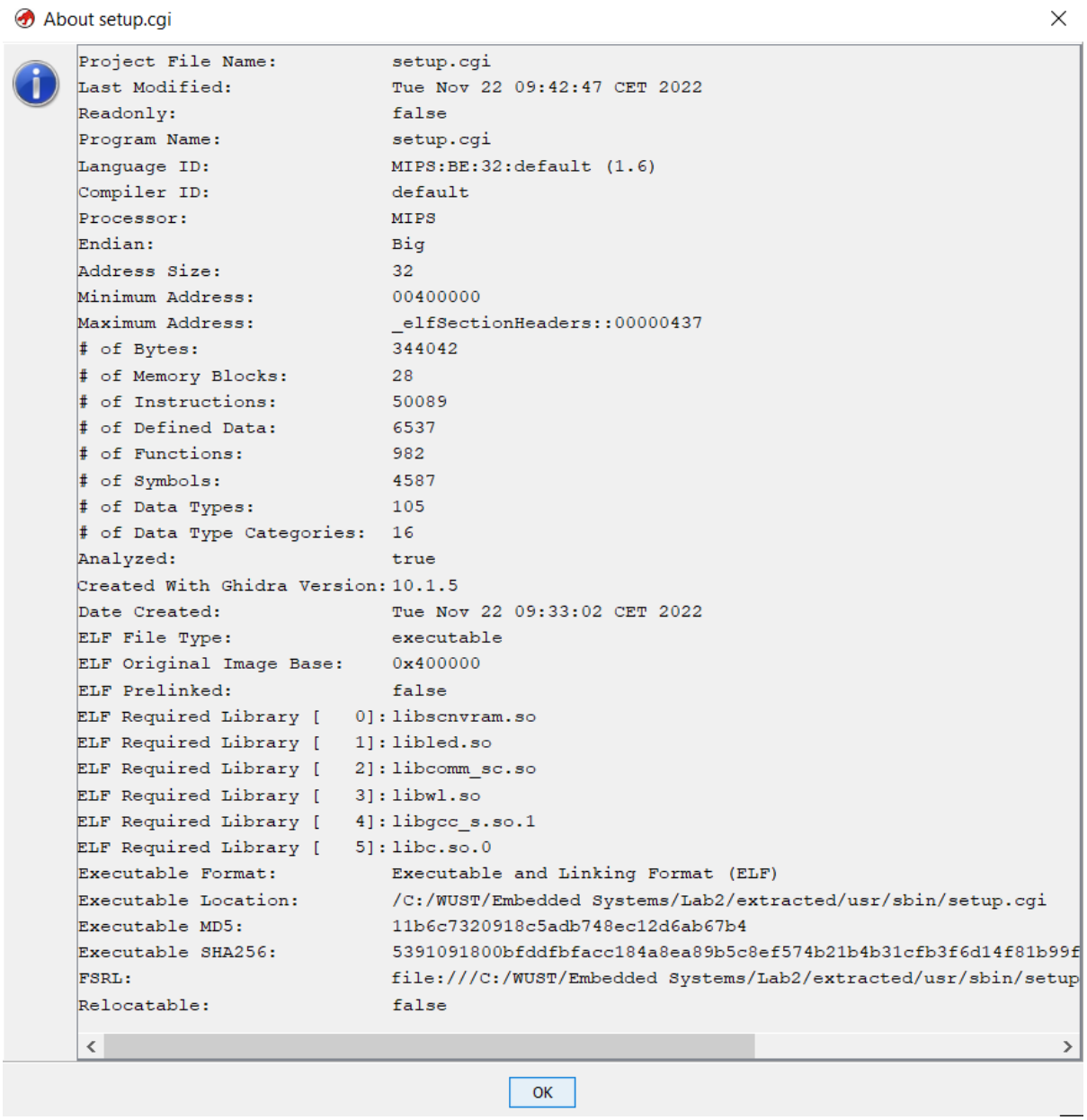


Figure 1.12: Importing a file in Ghidra

Figure 1.13 shows how the project looks, after including other required imports, all found within the firmware file.

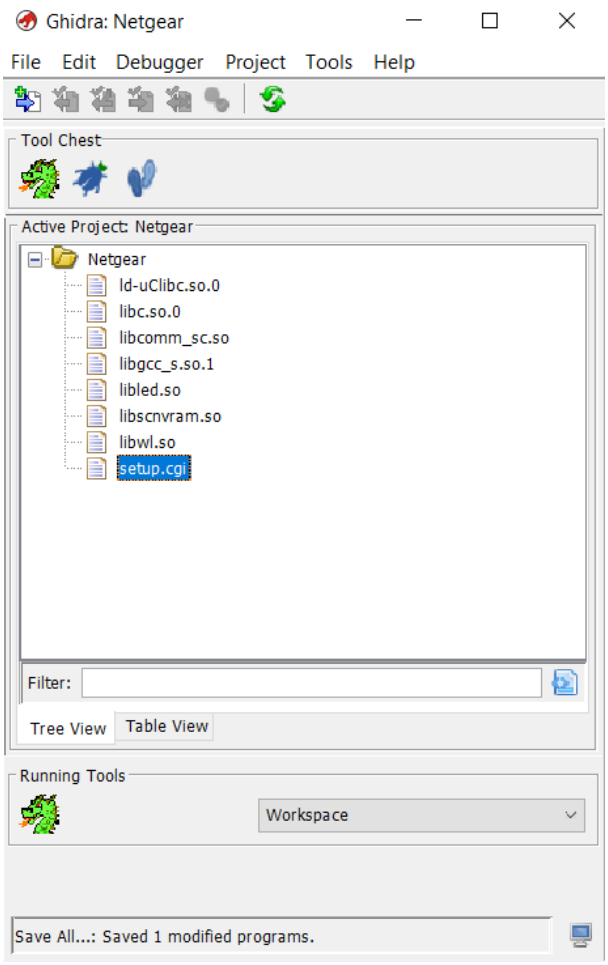


Figure 1.13: Project with imports

1.11.2 Using the Code Browser

Using keywords such as `main` and `entry` to search via the symbol tree, we can observe a couple of initialization calls to specific methods, such as:

- **unnamed first call:** This function checks for `/tmp/wan_uptime` to possibly get the wide area network uptime previously saved in a file in this location; then check for the `wan_ifname` value inside non-volatile memory, and unless it is set to `10.64.64.64` (likely a reserved IP) returns that. Using Ghidra's function naming functionality, I have just named this function `hakan_getWanUptime` accordingly.

```
char * hakan_getWanUptime(void)
{
  FILE *__stream;
  int iVar1;
  char *pcVar2;
  int iVar3;
  char acStack_30 [20];
```

```
  in_addr local_1c;

  __stream = fopen("/tmp/wan_uptime","r");
  if (__stream != (FILE *)0x0) {
    fclose(__stream);
    iVar1 = socket(2,3,0xff);
    if (iVar1 == -1) {
      perror("socket creating failed");
    }
    else {
      pcVar2 = (char *)nvram_get("wan_ifname");
      if (pcVar2 == (char *)0x0) {
        pcVar2 = "";
      }
      strcpy(acStack_30,pcVar2);
      iVar3 = ioctl(iVar1,0x8915,acStack_30);
      if (iVar3 == 0) {
        pcVar2 = inet_ntoa(local_1c);
        pcVar2 = strdup(pcVar2);
        close(iVar1);
        iVar1 = strcmp(pcVar2,"10.64.64.64");
        if (iVar1 != 0) {
          return pcVar2;
        }
        return (char *)0x0;
      }
      close(iVar1);
    }
  }
  return (char *)0x0;
}
```

- **dnslookup:** Checks if the entered DNS address has the format `number.number.number.number`. One interesting thing to notice in this function is how one functionality is passing the control to another, in other words, how page navigation is done:

  ```
  [..]
  // param_1 is passed as argument
  uVar3 = find_val(param_1,"next_file");
  html_parser(uVar3,param_1,&PTR_s_post_par_10001550);
  return 0;
  ```

  This is the basis for some of the publicly known security vulnerability exploits, such as the one we discussed at section 1.10.1.

- **set_c4_rsv_ip:** This function gets many values from the non-volatile storage using the nvram_get call, these variables are namely `dhcp_reserved`, `lan_ipaddr`, `dhcp_start_ip`, `dhcp_end_ip`, `lan_netmask`, all self-explanatory.

- **detect_proc:** This function seems to be for adding WPS clients, besides trying to read a file called `/tmp/porc_state`. It is also obvious how the navigation is done

based on the result of operations:

```
undefined4 detect_proc(void) {
 [...]
 if (iVar4 == 0x34) {
   iVar4 = strcmp(pcVar3,"client-pbc");
   if (iVar4 == 0) {
     pcVar3 = "setup.cgi?next_file=WPS_Add_Client_FAIL_Timerout
      .htm";
   }
    else {
     pcVar3 = "setup.cgi?next_file=WPS_Add_Client_FAIL.htm";
    }
  }
  [...]
  pcVar3 = "setup.cgi?next_file=WPS_Add_Client_OK.htm";
  [...]
  location(pcVar3);
  nvram_set("pro_count",&DAT_0044144c);
  return 0;
}
```

Thus another function of interest is `location` which seems to be just printing passed-in location parameter.

```
void location(undefined4 param_1)
{
  /* WARNING: Treating indirect jump as call */
  printf("Location: %s\n\n",param_1);
  return;
}
```

At least the usage seems safe from the format string attack.

- **COMMAND:** A very interesting piece of code is this function, which is called whenever a shell command needs to be executed. The command seems to be written to a file, namely `/etc/cmd_agent` before being executed:

```
bool COMMAND(char *param_1,undefined4 param_2,undefined4 param_3,
undefined4 param_4)
{
  FILE *__s;
  undefined4 local_res4;
  undefined4 local_res8;
  undefined4 local_resc;
  char local_408;
  undefined auStack_407 [1023];

  local_408 = '\0';
  local_res4 = param_2;
  local_res8 = param_3;
  local_resc = param_4;
```

```
  memset(auStack_407,0,0x3ff);
  vsnprintf(&local_408,0x400,param_1,&local_res4);
  __s = fopen("/etc/cmd_agent","w");
  if (__s != (FILE *)0x0) {
    fwrite(&local_408,0x400,1,__s);
    fclose(__s);
    usleep(1);
  }
  return __s == (FILE *)0x0;
}
```

There seem to be 109 calls to this function, which we can list via right-clicking the function name and selecting `References > Find References to COMMAND`, displayed in figure 1.14.

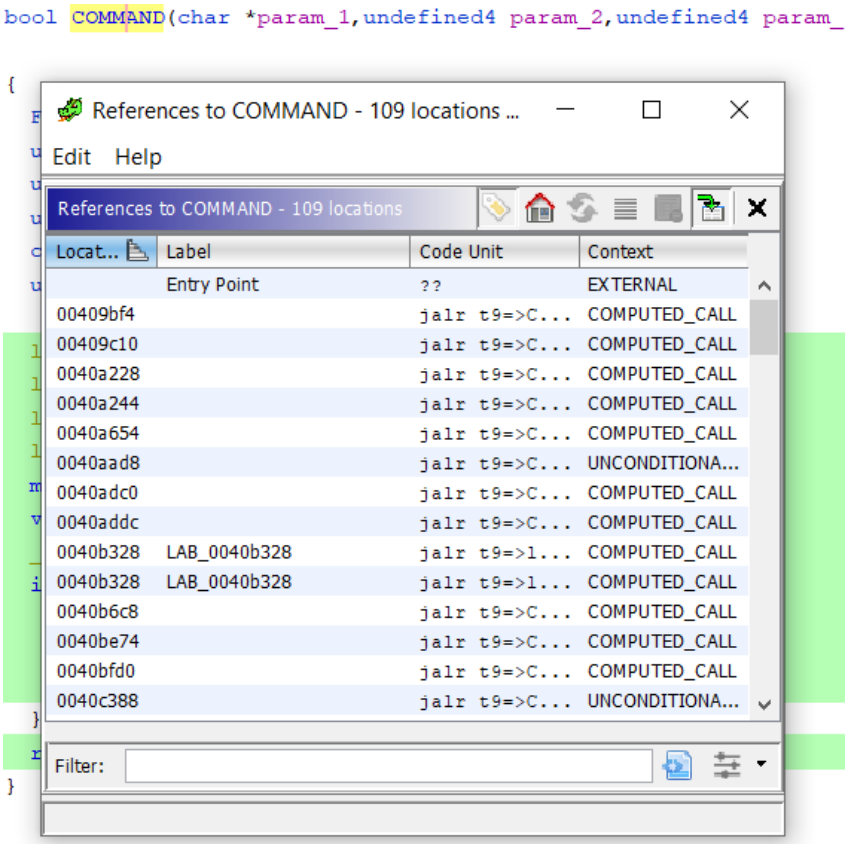


Figure 1.14: Calls to the COMMAND function

- **fwcheck:** This function is responsible for checking for a new firmware version on the internet. A portion of the code which is of interest is as follows:

```
[...]
pcVar1 = "updates1.netgear.com/dgn1000/ww/";
iVar2 = download_file("updates1.netgear.com/dgn1000/ww/",
 "fileinfo.txt","/tmp/LastestVersion");
if (iVar2 == -1) {
    pcVar1 = "updates2.netgear.com/dgn1000/ww/";
    iVar2 = download_file("updates2.netgear.com/dgn1000/ww/",
```

```
      "fileinfo.txt","/tmp/LastestVersion" );
    if (iVar2 != -1) goto LAB_004126dc;
    pcVar1 = "updates3.netgear.com/dgn1000/ww/";
    iVar2 = download_file("updates3.netgear.com/dgn1000/ww/",
     "fileinfo.txt","/tmp/LastestVersion" );
    if ((iVar2 != -1) &&
     (iVar2 = get_down_para("/tmp/LastestVersion", server_para),
     iVar2 == 0)) {
      iVar2 = check_need_upgrade(server_para);
      goto joined_r0x00412790;
    }
}
[...]
```

The URL **updates1.netgear.com/dgn1000/ww/fileinfo.txt** leads to a text file with the content **NETGEAR** only, which suggests the firm stopped hosting any firmware information at this location at some point. (second and third URLs found in the code above do not return any data) However, they seem to have moved the firmware location to a different subdomain, at http.fw.updates1.netgear.com/dgn1000 /ww/fileinfo.txt which returns this plaintext data:

```
[Major1]
file=DGN1000-V1.1.00.50WW.img
md5=1FA7BC0D92A0A3A22972B9F188EBC08A
size=4194304
c1=<MSG100>
o45=<MSG145>
o49=<MSG149>
o50=<MSG150>
```

Our router does not fetch this page, but it is possible the company released a newer firmware with correct URLs. Still, the fact that the URL was changed renders auto-upgrade capabilities for initial versions obsolete.
From another point of view, since we can see the hard-coded URLs the device is fetching, another attack vector could be registering a device with the name i.e. **updates1.netgear.com**, hosting a fake API under path **/dgn1000/ww/** and forward the traffic into this fake server, possibly hosting a legitimate-looking firmware file and let the unsuspecting device download and install it. This can be made easier using another piece of information explained in the next section.

1.12 Open-Source Intelligence

We have come across a page titled **NETGEAR Open Source Code for Programmers (GPL)** found here. We downloaded the corresponding archive file for our router and the contents seemed to contain the firmware itself, source code for the third party libraries used by the device firmware including **busybox v1.13.1**, **gpio** and **nvram**.

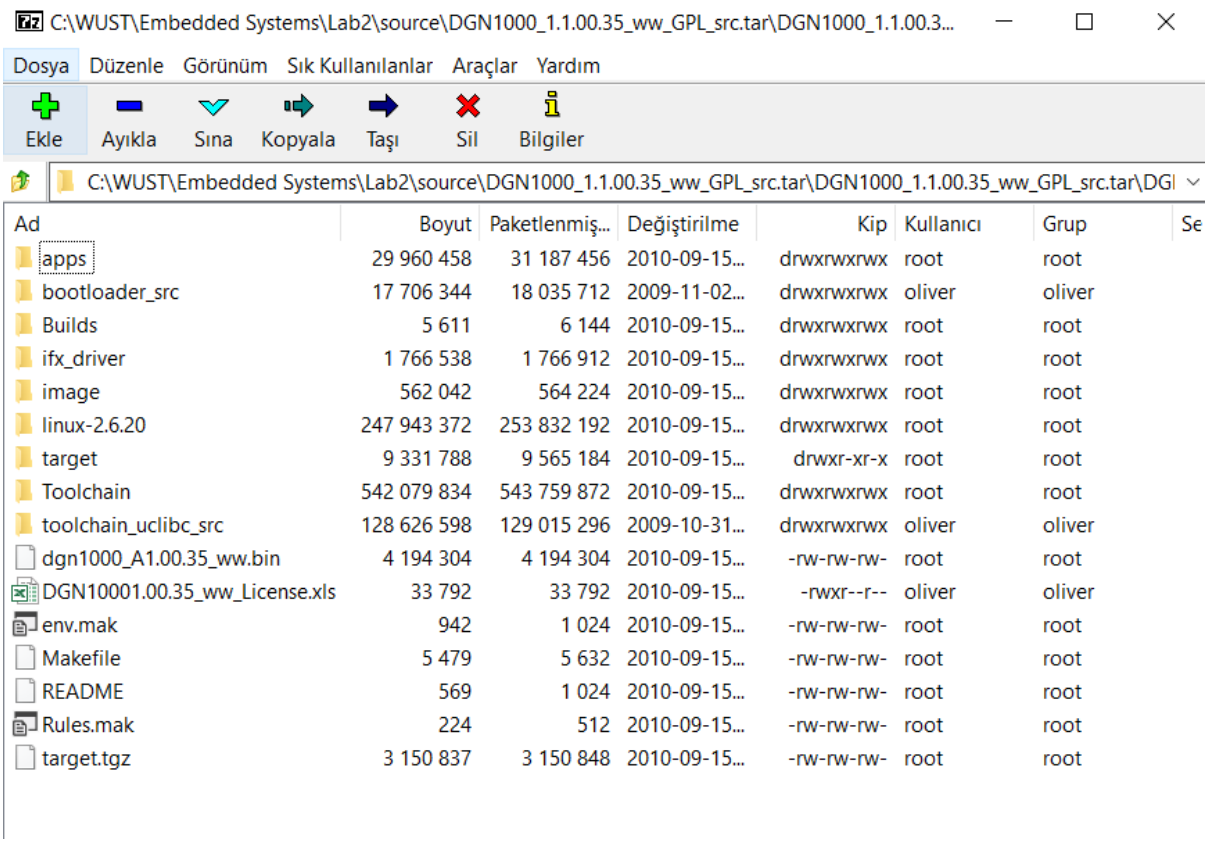


Figure 1.15: Contents of the file for our router

The folder `target` contains the actual file system inside the firmware.

Hence, although we have not attempted this, tweaking the source code for such critical third party library apps, i.e. in a way that would leak sensitive information, then building a malicious firmware file using these, and hosting it in the local network could be another attack vector.

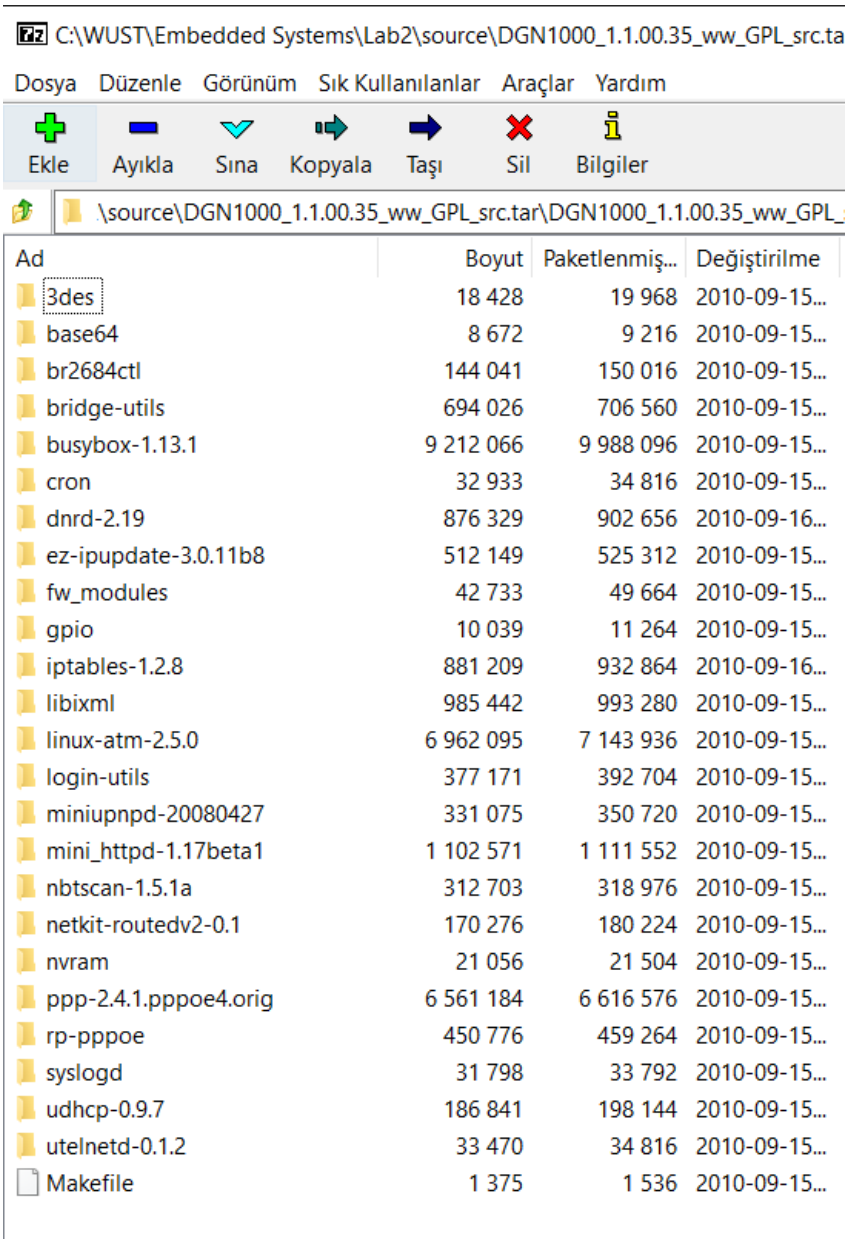


Figure 1.16: Source code for 3rd party apps

From the list of third party libraries, **mini httpd v1.17** piqued our interest, as we found a very critical vulnerability that is present in all versions up to v1.30 as disclosed on the library's website at acme.com/software/mini_httpd:

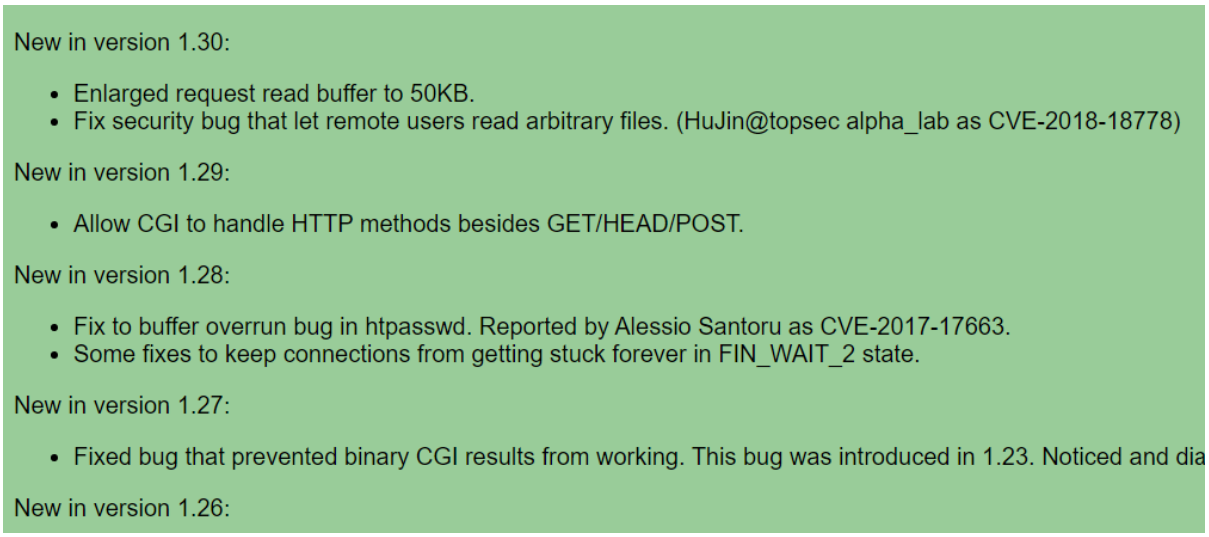


Figure 1.17: Disclosure regarding CVE-2018-18778

Although the exploit is not made publicly available, combining information we gather from various resources, it seems to be some kind of **path-traversal** vulnerability discovered in 2018, which is made possible due to the library not handling **GET requests with an empty header** properly, resulting in remote users being able to **read arbitrary files**.

2 | Final remarks

2.1 List 3

We were able to perform many actions on the NETGEAR DGN1000 router that were unintended by the developers. We gained much information about the device configuration, firmware, working principles, chipset, and finally exploits. Considering that the device is old, cheap, and still accessible without any problem, it is nice to know how much information one can learn simply by playing with the device. Furthermore, websites dedicated to hardware analysis helped to identify components inside the device. The device picked by us was friendly in a reverse-engineering sense and showed very little resistance. Also as not the newest, many websites and blog posts proved very useful. So getting shell injection possibility was as easy as browsing CVE website.

Bibliography

[1] *Wireless-N 150 ADSL2+ Modem Router*, https://fccid.io/PY309300114, Accessed: 13-11, 2022.