Fuzz Testing of a Wireless Residential Gateway

Noah Holmdin[1] and Martin Gilje Jaatun[2]

[1] NTNU, Gjøvik, Norway
[2] SINTEF Digital, Trondheim, Norway

Abstract. The rise of cyber-attacks against the ever-expanding network connectivity has resulted in a need for conducting security assessments in home gateway devices, which serve as junctures between private and public networks. Fuzzing, a method where invalid, random, or unexpected data is injected into a system, has emerged as a potential candidate for such assessments. An important aspect of conducting fuzzing is the implementation of monitoring tools to capture data that causes the target to behave unexpectedly. This study found that both a process monitor and a network monitor are essential for overseeing the fuzzing session. The process monitor tracks the status of the target process, while the network monitor captures network traffic between fuzzer and target. The findings demonstrate that fuzzing is an effective tool for conducting security assessments of home gateway devices.

Keywords: Security · Testing · Fuzzing

1 Introduction

In the era of ever-expanding connectivity, the role of the home gateway has become increasingly prominent. These devices represent the first line of defense against cyber threats. According to Zscaler [24], malware attacks against *Internet of Things* (IoT) devices had grown 400% between 2022 and the first half of 2023, which highlights the importance of keeping networks secure. The telecom provider (TelCo) needs to be able to assess the security on their clients' home gateway models, which are often provided by third-party manufacturers.

This paper studies the practice of security assessment through an approach known as *Fuzzing* [12] - a method where invalid, random or unexpected data gets injected into a system with the aim of uncovering vulnerabilities. Specifically, we study the network fuzzing of home gateway devices, involving generation and transmission of mutated data.

2 Background

Liang et al. [10] present an overview on the effectiveness, obstacles and future directions of fuzzing techniques. They present reviews of different fuzzing tools and what type of flaws they can detect. They mention the network protocol fuzzer *Sulley* [1] and its flexibility in giving the user the ability to generate test cases based on defined *blocks*, and that *boofuzz* [17] is a fuzzing tool based on Sulley, as the aforementioned is no longer maintained [10]. Mantu et al. [11] highlight the challenges of evaluating fuzzing tools, due to factors including limitations on disclosing detected vulnerabilities. They conclude that fuzzing has become the most utilized automatic testing method.

In addition to fuzzing, several other security testing techniques are commonly used, such as *vulnerability scanning*, *static code analysis*, and *manual penetration testing*. Vulnerability scanning tools focus on identifying known vulnerabilities based on signatures, which limits their effectiveness in detecting unknown vulnerabilities [7]. Static code analysis reviews source code without execution and, while scalable, often misses vulnerabilities and generates false positives [6].

Due to the growth of fuzzing, new tools are constantly developed and designed for certain types of targets. *IoTHunter* [23] is applicable for fuzzing *Internet of Things* (IoT) firmware. They evaluate its code coverage, which is shown to outperform boofuzz, and present that the tools managed to find five vulnerabilities on a router device, two by fuzzing the *File Transfer Protocol*(FTP) one the *Server Message Block Protocol*(SMB) and two by fuzzing the *Simple Network Management Protocol*(SNMP) [23]. Another tool, specifically designed for testing routers is the *RPCFuzzer* [21]. This tool generates tests by a presented mathematical model that can generate a wide range of inputs. An experiment is conducted to test RPCFuzzer's effectiveness by comparing it with other fuzzing tools. The experiment targets two router set ups with the SNMP service being the attack vector. The RPCFuzzer outperform Sulley by finding eight bugs where Sulley found none[21]. Due to the test generation model a total of more than 2.1 million test cases were generated in comparison to Sulley's ∼520000 [21], however this also drastically increased its run time, highlighting the trade-off between test generation and time consumption. These studies highlight the different targets viable for fuzzing within gateway devices. This is further shown by Li et al. [9] who evaluate their fuzzer based on the *Semi-valid Fuzzing Test Cases Generator* (SFTCG) model. They target the *Internet Control Message Protocol* (ICMP) service of a Cisco router and found two Denial of Service vulnerabilities.

Many fuzzing related studies follow the theme of developing and evaluating a tool. Most have the fuzzer generate inputs based on patterns and known vulnerabilities. A different approach is shown in the work of Wen et al. [22], in which their tool makes use of a middleman set up to capture valid network traffic, and then deriving seeds from the traffic to generate test cases.

An important and challenging aspect of fuzz testing is having the ability to capture unexpected events during the fuzzing session, such as knowing whether the target has crashed, or encountered an exception. Wang et al. [21] use a monitoring routine including manual inspection of a router's system log, tracking the CPU utilization and using a debugger attached to the targeted process. They conclude that their setup solves monitoring problems of previous studies targeting router devices. Similarly, Li at al. [9] use a debugger to monitor the target process, and like Wang et al. [21], they also use SNMP to receive information about the target system throughout the fuzzing sessions.

3 Context

Home gateway devices are third party products, meaning that the TelCos have limited control over the software and hardware of the devices, and are reliant on the third party for updates and patches. This has resulted in a need for security assessment measures that would enable the TelCo to perform security testing.

For the purpose of this research, the *threat actor* - an individual exploiting vulnerabilities for cyber-attacks[8] - can be situated both within and outside of the gateway subnet and is limited to using attack vectors accessible through network communication. The threat actor does not have access to the gateway's web-interface and can thereby not modify configurations or remove firewall rules. The gateway device utilized in this study is provided by a TelCo and runs a customized version of the Linux distribution *OpenWRT*.

A vulnerability represents a weakness in a system or its design that can be exploited by a threat actor [15]. These can exist in software, hardware or in practical procedures, and can be targeted by various cyberattacks. The following are some common types of attacks that exploit vulnerabilities.

– **Denial of Service (DoS)** - typically overwhelms targets
– **Ping of Death (PoD)** - crashing vulnerable servers with pings
– **Buffer Overflow** - DoS or remote code execution
– **SQL injection** - database compromise
– **Cross-Site Scripting (XSS)** - injecting malicious scripts
– **DNS Poisoning** - a.k.a. DNS spoofing

4 Fuzzing a Virtual Environment

The first cycle of the study is centered around the development of a fuzzing environment, finding viable targets and setting up a thorough monitoring setup.

4.1 Diagnosing

To evaluate the use of fuzzing as a security assessment tool for gateways, it was critical to understand the architecture of the fuzzing framework and how to choose fuzzing targets. The tool had to be adept at generating network payloads with input mutations that cover a wide range of potential vulnerabilities. It was also important that sufficient documentation was available.The target machine had to be set up with a communication channel to the fuzzer as well as simple networking services, mimicking common services of a gateway device. It was also important to have the ability to monitor target processes, to see how they respond to mutated and unexpected data. In the initial cycle of the study it was deemed beneficial to set up a target that could be configured and monitored easily as the fuzzing scripts were continuously modified and improved.

4.2 Action Planning

The fuzzer *boofuzz* version 0.4.2 [17] was selected due to its comprehensive documentation and robust feature set. *boofuzz* excels in generating a wide range of data types, including strings, bit and byte streams, integers, and floats. It mutates data based on predefined inputs and patterns that are known to potentially cause vulnerabilities. The tool is highly flexible and can be used to mimic various types of network data and protocols. It can be configured to be used as both a black-box and grey-box fuzzer, and its mutation techniques facilitate an extensive exploration of target states, with each defined field having a default value that is iteratively mutated.

When the fuzzer has sent all the mutated values of one field, the field will be set to its default value and the fuzzing will continue mutating the subsequent field. It is possible to set a max depth value when starting the session which controls how many fields will be fuzzed together. If set to one, the fuzzer will mutate fields separately. If set to two, it will mutate each combination of two fuzzable fields together. If set to maximum depth, all fields are mutated together.

Although a deeper maximum depth facilitates greater code coverage, it also results in an exponential increase in the number of test cases, extending the time required for testing. We thus decided to fuzz with a maximum depth of 1. *boofuzz* also includes monitoring tools, including the `ProcessMonitor`, which provides crash feedback when attached to a process, and the `CallbackMonitor`, which allows custom functions to be triggered after each test case.

The fuzzer host operated on a mid-high range laptop equipped with 16GB of RAM and an i7-8750H 2.20GHz CPU [16], running Ubuntu 23.10, providing a platform capable of performing fuzzing without much performance constraints.

As the target needed to mimic how a gateway device operates, the Linux distribution OpenWRT was chosen, for its simplicity and capability to emulate the functionalities of a gateway device. Also, OpenWRT components have been targeted in several previous fuzzing studies [20,5], making it a suitable firmware to run on the target machine. Initially, version 18.06.4 was considered as several vulnerabilities had been discovered on it. This was later changed to version 22.03, due to it supporting python 3.9, a requirement for boofuzz. To enable easy configuration and management, the OpenWRT system was used through a *Virtual Machine* (VM) with *Virtual Box* [2]. The targeted processes were limited to protocol services that were set to listen on designated ports, and in the case of this virtual OpenWRT system, these were HTTP, SSH and DNS, respectively managed by the services: `uhttpd`, `dropbear` and `dnsmasq`. The default field values would be based on valid HTTP, DNS and SSH requests from the host to the target, based on traffic observation between the host and target machine.

To analyze the fuzzing session the plan was to set up a thorough logging and monitoring configuration. This also required a distinction to be made on what was to be counted as an *unexpected response* from the target;in our case, an unexpected response occurs when either of the following criteria are met:

1. Target machine crash.
2. Target process crash.

3. Target handling malformed packet as valid.
4. Target resource utilization abnormality, caused by the malformed packets.

Each of these criteria needed to be monitored and as fuzzing is time demanding it would be beneficial for the system to do it automatically and report when abnormalities occur. To detect whether the target machine had crashed, OpenWRT's kernel log was manually inspected, as done in [21]. Process crashes can be detected by attaching debuggers to the process IDs at the target machine, as done in [21] and [13], and as boofuzz includes the `ProcessMonitor` object, it could be utilized in the fuzzing harnesses and connect to an instance of boofuzz's `process_monitor.py` script, situated at the target machine. The object and script communicates through a *Remote Procedure Call* (RPC) server, enabling the fuzzer to run commands on the target machine and the target to provide feedback such as crashes to the fuzzer. Initially, the plan was to use the *vtrace debugger* introduced in [3], as it provides deep synopsis on exceptions and crashes. Due to an error in OpenWRT concerning the file pointer variable *lseek64*, this was not possible. Instead, the plan was changed to use boofuzz's *simple debugger*, which is able to provide feedback when process crashes with their error codes and correlate what test case caused the crash. A target treating malformed packets as valid is more difficult to track, due to the amount of data being sent and the difficulty of understanding exactly what requests should result in certain responses. For this, a decision was made to set up a network monitor, by using *tcpdump* to capture the fuzzing sessions network traffic and save it to a `.pcap` file, and subsequently use *Wireshark* to manually analyze the file. The analysis was done with an emphasis on finding patterns like: abnormal responses, multiple responses to single request, requests with no corresponding response and responses to request that do not adhere to protocol standards. Monitoring the resource utilization could be done at the target machine, with the software *top* for CPU utilization and *free* for memory utilization.

If an unexpected response occurred, the corresponding test case was analyzed with the help of the established monitoring and logging methods. All the mentioned modules resulted in a planned environment illustrated in Figure 1.

4.3 Action Taking

This section details the steps involved in establishing the virtual environment and conducting fuzz testing on the targeted system. The section covers the setup of the target system, the configuration of the fuzzing harness, the establishment of monitoring protocols, and the actual execution of the fuzz tests.

Setting up the target environment
Setting up the fuzzing target involved downloading and setting up the virtual machine running OpenWRT and configuring it with networking capabilities to allow for communication with the host. The chosen OpenWRT release had an ext4 file-system with the x86-64 instruction set. It was downloaded as an `.img` file which is not supported by the virtualization software `Virtual Box`, the img

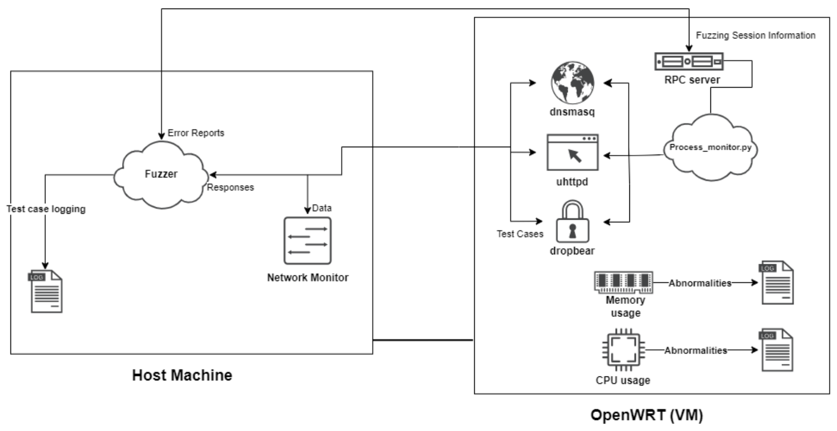


Fig. 1. Virtual Environment

file was therefore converted into a *Virtual Disk Infrastructure* (`.vdi`) file using the `VBoxManage` software's `concertfromraw` tool[3]. The `.vdi` file was then used to create a new virtual machine with an existing virtual disk with OpenWRT on it. The VM was allocated 256MB RAM, one CPU core and 255MB of disk space. The network set up is configured in accordance to the *OpenWrt on VirtualBox HowTo* guide on OpenWRT's wiki [2]. This provided the VM with NAT capabilities and an IP address capable of communicating with the host through Virtual Box's *host-only adapter* network settings. The subsequent step of setting up the VM included extending the root partition of the virtual, as the virtual disk was to small to contain the necessary software for monitoring the fuzzing sessions. Firstly, the `VBoxManage` command-line `modifyhd` is executed in the host environment to extend the virtual disk size to 2GB[4]. Secondly, the additional space is allocated to the root partition in accordance with a blog post posted on OpenWRT's forum [14], using tools such as *fdisk*. This resulted in a target environment capable of communicating and with more then enough memory to set up monitor tools.

Setting up the host environment

The setup of the host environment necessitates boofuzz, which itself depends on Python (version 3.9 or later) for execution and pip for installation. To safeguard the host system from potential disruptions caused by fuzzing activities, a decision was made to encapsulate the environment within a Docker container. By containerizing the fuzzing environment, Docker ensures that boofuzz operates under the same conditions regardless of the underlying host system. Given the choice of targeting three protocols, multiple scripts, operating as harnesses were developed

[3] `VBoxManage convertfromraw openwrt-23.05.3-x86-64-generic-ext4-combined.img openwrt.vdi`

[4] `VBoxManage modifyhd openwrt.vdi -resize 2048`

for use within the Docker environment. To facilitate switching between the fuzzing harnesses, the shell script `entrypoint.sh` was implemented and used in a Dockerfile, configured to deploy a container equipped with the Python libraries boofuzz, scapy, and requests.

Configuring the fuzzing harnesses
The fuzzing harnesses consisted of three python scripts, `http-cycle-1.py`[5], `ssh-cycle-1.py` and `dns-cycle-1.py`, each configured to fuzz various fields within their respective protocol, while still adhering to the protocol structure definitions. These harnesses were responsible for generating data in a structure following defined blocks, sending the data and maintaining the connection with the target.

The data fuzzed in `http-cycle-1.py` is shown in Figure 2 and consisted of multiple fields with different boofuzz mutation types. It ranged from the HTTP methods, URI with string, delimiters and various headers fields with delimiters, strings, integers and floats. This resulted in a robust HTTP fuzzer sending a large amount of malformed HTTP packets for the target to parse and process.

The DNS fuzzer mutates a large portion of the data in a DNS request, as shown in Figure 4. All header fields are fuzzed with binary values and the query fields is fuzzed with string values for the query name and binary values for type and class. Depending on the questions header value, the query fields were dynamically iterated to ensure adherence to standards. The structure of how the harness would generate payloads was based on a valid DNS request sent from the host machine to the target, using *nslookup*.

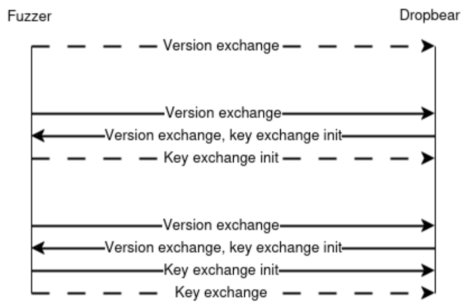
The SSH fuzzer mutates the three initial messages of setting up a SSH connection. Firstly, the protocol version exchange is sent with mutated data being generated for different parts of the request. Then, the protocol version exchange is sent with valid data and when the target responds the fuzzer sends fuzzed key exchange initialization messages, where each of the offered encryption algorithms are strings that get fuzzed. Lastly, the first two messages were sent with valid data followed by a key exchange message with mutated binary public keys. The protocol specification followed by the fuzzer was inspired from [18] and adapted by mimicking a valid SSH initialization process between the host and target.

Setting up monitoring and logging methods
The fuzzing harnesses were set up to log information from the output of the session into the console and a text file. Additionally, for the HTTP fuzzing script a `post_case_callback()` function parsed the received status code and appended the request and response in clear text to a file named in coherence with the status code, for instance `200.txt`. To monitor network traffic the packet capture software `tcpdump` was used inside of the fuzzing scripts by using the python library `subprocesses`. To save the session's packet trace to a `.pcap` file, the tcpdump command was used together with the -w flag, additionally the

[5] `https://github.com/Tossoen/fuzzing_scripts_boofuzz`

```
Mutated with the following methods: ("GET", "HEAD",
"POST","OPTIONS", "DELETE", "MERGE")
Mutated with integer values
Mutated with string values
Mutated with float values
Mutated with delimiter characters and strings
GET' '/index.html' 'HTTP/1.1
Host:' '{target_ip}
Connection:' 'keep-alive
User-agent:' 'fuzzer
Accept:' 'text/xhtml+xml,application/xml;q=0.9,*/*;q=0=0.8
...
```

Fig. 2. HTTP Test Case Generation



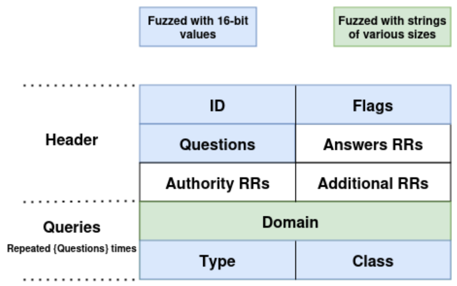

Fuzzed with 16-bit values
Fuzzed with strings of various sizes
Header
ID
Flags
Questions
Answers RRs
Authority RRs
Additional RRs
Queries
Repeated (Questions) times
Domain
Type
Class

Fig. 3. Fuzzing the SSH Initialization Phase

Fig. 4. DNS Test Case Generation

command was set up so that only traffic through the port corresponding with the targeted protocols were captured. This configuration made it possible to analyze the network traffic manually with the use of `wireshark`.

The next step was to set up the process monitoring on the target side. When installing boofuzz through pip, the installation does not come with the aforementioned `process_monitor.py` script, therefore the script was installed from the boofuzz git repository on the host machine and transferred to the target machine with the *secure copy protocol* (SCP). When running the script, it initializes and sets up a listening port and waits for a connection to be made from the host. To set up the connection the fuzzing scripts used a `ProcessMonitor` object and specified what process on the target should be monitored along with start and stop commands for the process. This provided the target with the ability to log unexpected crashes and report it to the host.

4.4 Evaluation & Learning

The fuzzing session targeting the uhttpd service revealed that the service produced unexpected responses, including 200 OK for malformed requests and a request-response loop bug caused by specific patterns in the URI field. The session also revealed that packets exceeding the byte limit were incorrectly processed, with

Table 1. Results of The Conducted Fuzzing in Cycle 1

	HTTP	**DNS**	**SSH**
Test cases executed (amount)	30 450	5 312	32 301
Session run time (minutes)	154	245	68
Unexpected response encountered (yes/no)	Yes	No	Yes
Process crash encountered (yes/no)	No	No	Yes
Machine crash encountered (yes/no)	No	No	No
CPU / memory abnormality occurred (yes/no)	No	No	No

the server returning both 403 Forbidden and 413 Payload Too Large instead of consistently responding with the 413 code, indicating a flaw in how the service handles such requests. The DNS fuzzing session showed that the dnsmasq service effectively handled malformed requests by refusing them or dropping them. Additionally the session highlighted areas of improvements in the generation of DNS queries to further the amount of states explored. The SSH session mostly resulted in connection refusals, but also a false-positive crash likely caused by network issues, which highlighted the importance of monitoring tools capable of providing accurate synopsis of crashes.

5 Fuzzing the Physical Gateway Device

The second cycle of the study is centered around performing fuzzing against a physical gateway device.

5.1 Diagnosing

To determine whether fuzzing is a good security assessment tool for home gateways, the fuzzing harness had to be adept at creating an array of malformed data and the gateway had to be able to run monitoring tools. The scripts used in cycle one were good starting points, but there was definitely room for improvements. A physical home gateway device could have additional services that are listening to ports compared to OpenWRT. This meant that new protocols, services or other software may be viable for fuzzing, which would benefit the evaluation since it would showcase the fuzzing can target more modules and services within the gateway device. As the monitor setup proved efficient, the same configuration needed to be set up in the new environment.

5.2 Action Planning

The gateway device was provided by the TelCo; it ran on an OpenWRT-based firmware, and was accessible through SSH. The first consideration made in the second cycle was which protocol services to target. The gateway had services handling the three protocols targeted in the first cycle, and thus deemed a good fit. The decision was made to reconfigure the HTTP and DNS harnesses. The

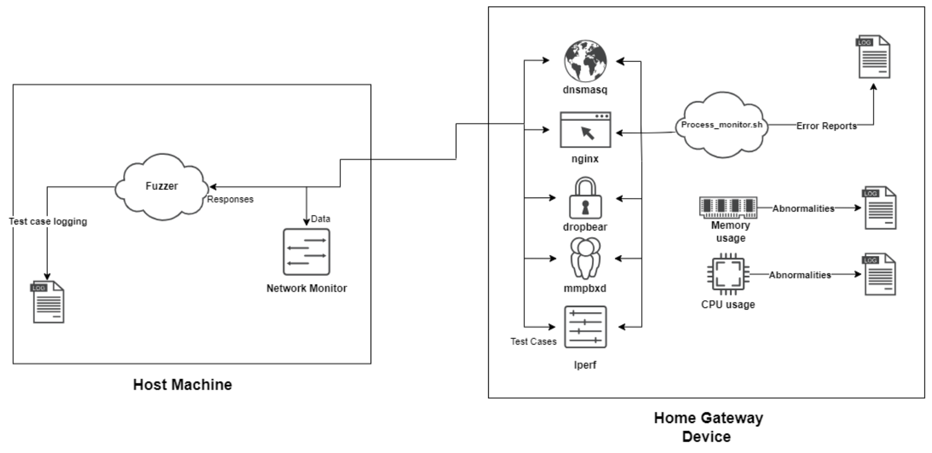


Fig. 5. Cycle 2 Experimental Environment

SSH script would also be used but not modified, as the SSH initialisation is complex and the likelihood of successfully replicating the key exchange init was deemed small. Furthermore, other possible targets on the gateway were identified, including a SIP service and the *iperf* software. Fuzzing of SIP interfaces has been explored in [19] and several interesting findings were discovered; SIP is not as commonly used as HTTP, SSH and DNS, making it interesting to study with fuzzing. Iperf is especially interesting; unlike the other targets, iperf is not a protocol service but a bandwidth measuring tool, widening the types of targeted processes in the study. Another takeaway from the fuzzing sessions in cycle 1 was that a deeper max depth for the mutations could prove beneficial, as it would increase the code coverage and exploration of the target states.

Due to issues with Gateway's package manager, the process monitor utilised in cycle 1 could not be reused in this cycle. Therefore, there was a need to develop a custom process monitor using the preinstalled shell script language ash. Since direct communication between the process monitor and the fuzzing harness was not feasible, an alternative approach using timestamps was planned, allowing for manual correlation between crash logs and test case timestamps by inspecting captured traffic.

5.3 Action Taking

Setting up the Target Gateway
An Ethernet link operating at maximum 1000Mb/s was set up between the fuzzer laptop and gateway device, this created a communication channel between the devices through a shared subnet.

Modifying and Creating new Fuzzing Harnesses
The evaluation of the HTTP fuzzing harness utilized in the first cycle revealed

some potential areas of improvement. This led to a new harness being created[6] that generated test cases with fewer HTTP methods and fewer mutable fields. The URI was set to be fuzzed with strings, and the only header field was the user agent. Both of these fields were set to generate strings no longer that 75 characters, to lessen the amount of time it takes to send the test cases. An additional HTTP structure block was implemented in the harness, that only mutated body content of the HTTP messages, with the intent of seeing how the target process responds to fuzzed body data. For the DNS fuzzing harness[6], an *authority* section of the DNS request was added whenever the header field *authority* was set to 1, increasing the amount of fields fuzzed compared to the previous iteration of the harness. Additionally, a callback monitor was implemented within the DNS harness that parsed response codes from responses, and appended them into a text file. The modified DNS harness is demonstrated in Figure 6. Creating harnesses for the two new targeted services involved the following: The SIP fuzzing script[6] was created to mimic a SIP invitation message, with a focus on fuzzing header values. All headers were fuzzed with string values and the URI was fuzzed with strings for name and domain, and delimiters for the "@" and a body fields were added and fuzzed with strings. The default values for the harness were generated with the large language model *chatgpt*, due to difficulties in replicating an accurate SIP INVITE. The iperf harness generated streams of bytes in various lengths and types, by using four instances of boofuzz `s_bytes()` objects inside of a boofuzz block.

Setting up the Monitoring Configuration
Due to the gateway's memory being filled, a USB storage device formatted with an ext4 partition was mounted on the gateway system[7], and the mount was made persistent[8]. This provided storage for the monitoring tools and log files. The CPU and memory monitors were transferred to the gateway using SCP, and their thresholds for abnormalities were adjusted accordingly.

A custom process monitor, `process_monitor.sh`, was created using ash shell script. This script continuously monitored process IDs through the `check_processes()` function, that utilized `pgrep` based on the process name provided as an argument at script execution. If a process ID disappeared, indicating a crash, this event was logged. Additionally, in the event of a crash, the `check_kernel_log()` function was used to determine if new kernel events had been logged since the last check. If new entries were found, the latest 20 entries from the kernel log were logged. Each log entry was prefixed with the date and time.

5.4 Evaluation & Learning

The results of the fuzzing sessions against the physical home gateway device is summarized in Table 2.

[6] `https://github.com/Tossoen/fuzzing_scripts_boofuzz`
[7] `mkdir /mnt/usb && mount /dev/sda4 /mnt/usb`
[8] `echo '/dev/sda1 /mnt/usb ext4 defaults 0 0' » /etc/fstab'`

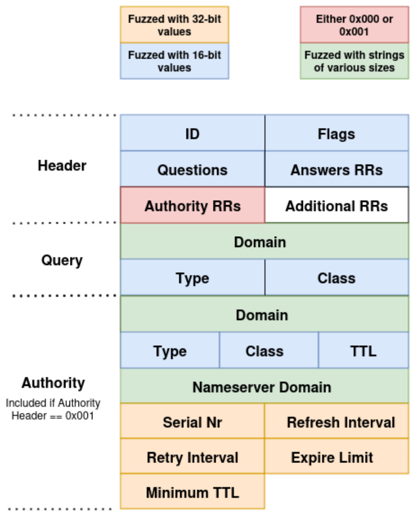


Fig. 6. Updated DNS Mutation Strategy

The HTTP fuzzing sessions ran for a total of 39398 test cases before being terminated, and is referred to as HTTP-2 in Table 2. The session was stopped because the focus on thoroughly exploring different fields together led to too many test cases. For example, each generated URI string was sent with approximately 700 mutations of the user-agent string. As a result, the session was concluded once sufficient data had been gathered. The results show that no requests were approved by the HTTP service, demonstrating its robustness. The status codes received were: 400, 401 and 405. Interestingly, the service never responded with the status code 413. This could possibly stem from the added limitation on the generated string value sizes, leading to packet sizes never exceeding the maximum allowed. To evaluate further, the HTTP harness used in the first cycle was reused (referred to as HTTP-1 in Table 2) and the session resulted in responses with status codes 400, 401, 405 and 414. 414 was returned when the URI extended a specific size, and instead of returning 413 when the request size extended the maximum, 400 was returned, and whereas it can be argued that this does not follow the HTTP standards[4], it still indicates that the requests are correctly not being processed. Out of the 30450 test cases, the HTTP service responded with 400 for 30004 of them.

The SSH fuzzing session resulted in no unexpected responses. Even though the gateway device ran the same SSH service as the virtual environment, no crash was encountered, hinting that the process crash in cycle 1 was caused by high traffic load or similar factors.

Similar results were seen from the DNS fuzzing session. The vast majority of responses from the server had a response code of 0001 or 0101, indicating that all malformed and invalid request were handled effectively. The callback function that parsed response codes simplified understanding the results of the

Table 2. Results from the Fuzzing Session Against the Gateway Device

	HTTP-2	**HTTP-1**	**DNS**	**SSH**	**SIP**	**iperf**
Test cases executed (amount)	39398	30450	19292	32301	27572	1052
Session run time (~ minutes)	54	30	322	33	15	4
Unexpected response encountered (yes/no)	No	No	No	No	No	Yes
Process crash encountered (yes/no)	No	No	No	No	No	Yes
Machine crash encountered (yes/no)	No	No	No	No	No	Yes
CPU / memory abnormality occurred (yes/no)	No	No	No	No	No	Yes

Table 3. Response Codes Received from DNS Fuzzing Session

Response code	**Amount**
0001	3731
0101	15501
0111	10
1111	50

fuzzing session. It provided valuable hints on what to look for when analyzing the monitored network traffic, similar to the HTTP status code parser. An issue with the DNS session was that between 4000 and 5000 test cases were duplicates. This was caused by the protocol specification implemented in the harness script, where the authority fields were only added when the authority header was set to 1. The implementation caused boofuzz to mutate each field within the authority section regardless of the value of the authority header. Even though the authority section was excluded when transmitting the packet, this led to unnecessary processing for both the host and target machines and generated unnecessary traffic through the Ethernet interface.

Fuzzing the SIP service resulted in responses with status codes 404 (NOT FOUND), 482 (DETECTED LOOP) and 400 (PARSE ERROR); all fuzzed inputs were handled efficiently and no unexpected responses were encountered, as shown in Table 2. An improvement of the SIP harness would have been to use valid default values for the SIP header and body fields.

The fuzzing session against the iperf software revealed issues, as it caused the process to crash, the gateway device to reboot and abnormalities in the CPU usage levels. The crash was consistently replicated on the gateway device, on the virtual environment used in cycle 1 and on a VM running an Ubuntu server. The cause was traced to an attempted write on an invalid memory address, resulting in a segmentation fault. The inputs that led to crashes followed a similar pattern, prompting the development of a payload capable of directly manipulating certain variables on the iperf server.

The payload began with a header value used to configure parameters for the iperf bandwidth measurement session, followed by data that could freely modify two variables of the iperf server. One variable tracks what specific port the software is listening on, the other specifies the allocated size of a buffer that is filled with data when the iperf server receives a packet. The segmentation fault occurred because the payload altered the buffer size to a value that exceeded the allocated buffer, resulting in buffer overflow and memory corruption.

By analyzing the issue with the memory checking tool valgrind, it was evident that the memory corruption resulted in memory leaks of varying degrees depending on the inserted buffer size value. This vulnerability is rooted in insecure design, where a seemingly intended design choice allows a client to transmit data that affects the functionality and operation of the software, leading to memory corruption and leaks on the machine running the iperf server. No previous reports of this vulnerability were found during research, it is therefore unlikely that traditional vulnerability scanning would have detected it.

The time required for fuzzing varied depending on the service being tested, ranging from just a few minutes for iperf to several hours for DNS. In practical implementations, the duration also depends on factors such as network link speed and the complexity of the service being tested. Most of the effort was initially spent on setting up the fuzzing environment, mainly due to the lack of documentation for the fuzzer and the need to understand the targeted protocols. However, once the necessary knowledge is acquired, the fuzzing setup process can be completed much more quickly. In comparison, static code analysis is generally faster, reviewing the source code without execution. However, it is limited to identifying code-level issues and cannot detect runtime vulnerabilities. Manual penetration testing can take days or weeks, depending on the system's complexity.

6 Conclusion

The primary objective of this study was to evaluate the feasibility of fuzz testing as a security assessment tool for home gateways, specifically within the context of a TelCo use case aiming to assess the security of their clients' gateway devices. Through an action research methodology conducted in two cycles, this study focused on creating and testing a fuzzing environment to identify suitable target interfaces, track session results, and discover potential vulnerabilities or bugs. In the first cycle, a virtual machine running the OpenWRT firmware with basic gateway capabilities was targeted to establish an initial fuzzing environment using the boofuzz framework. This cycle targeted OpenWRT's HTTP, SSH and DNS protocol services, and the results from this cycle informed improvements for the second cycle, which targeted a physical home gateway device and expanded the targets to include the SIP protocol service and iperf software. The findings indicate that fuzzing gateway devices enables automatic generation and transmission of thousands of test cases. Additionally, effective analysis and evaluation require a process monitor and a network monitor, along with monitoring resource utilization and system logs for other abnormalities. Through the conducted fuzzing, a bug was discovered in the HTTP service of OpenWRT, were a specific URI caused a request-response loop, and a buffer overflow vulnerability, causing the gateway to reboot was discovered in the iperf software.

These results highlight the potential of fuzz testing as a method for identifying vulnerabilities in home gateway devices. While not a definite solution, fuzzing is effective at discovering edge cases and unknown vulnerabilities that techniques such as vulnerability scanning would miss, as seen with the iperf vulnerability. The study also highlights the importance of continuous monitoring and the need for thorough testing strategies across various interfaces and aspects of home gateway devices.

Future research could expand on our results by conducting more cycles and exploring the target states further, with a focus on finding vulnerabilities and discussing their implications. Additionally, evaluating the potential of grey-box fuzzing, where test cases adapt based on responses from the gateway, could help detect more complex vulnerabilities that were not identified in this study.

Another promising area would be to compare different frameworks to see which are most effective in the context of home gateway testing. This, and fuzzing home gateways, is a relatively unexplored area and could lead to the development of more thorough guidelines on how TelCos can integrate fuzzing as a security assessment method to keep their clients' networks safe.

Acknowledgements

This work is based on the first author's BSc thesis at Linköping University, and has been supported by the European Commission through the Horizon Europe TELEMETRY project, grant number 101119747.

References

1. Amini, P., Portnoy, A.: Sulley fuzzing framework manual. `http://www.fuzzing.org/wp-content/SulleyManual.pdf`, accessed: 2024-04-10
2. atownlede: VirtualBox VM Setup Guide. OpenWrt Documentation (Oct 2023), `https://openwrt.org/docs/guide-user/virtualization/virtualbox-vm`, accessed: 2024-04-15
3. Cousineau, P., Lachine, B.: Enhancing boofuzz process monitoring for closed-source SCADA system fuzzing. In: 2023 IEEE International Systems Conference (SysCon). pp. 1–8. IEEE (2023)
4. Fielding, R.T., Nottingham, M., Reschke, J.: HTTP semantics. Internet Engineering Task Force RFC 9110 (2022), `https://httpwg.org/specs/rfc9110.html#overview.of.status.codes`, accessed: 2024-05-13
5. Gao, J., Xu, Y., Jiang, Y., Liu, Z., Chang, W., Jiao, X., Sun, J.: Em-fuzz: Augmented firmware fuzzing via memory checking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(11), 3420–3432 (2020)
6. Goseva-Popstojanova, K., Perhinschi, A.: On the capability of static code analysis to detect security vulnerabilities. Information and Software Technology **68**, 18–33 (2015)
7. Holm, H., Sommestad, T., Almroth, J., Persson, M.: A quantitative evaluation of vulnerability scanning. Information Management & Computer Security **19**(4), 231–247 (2011)

8. IBM: Threat actor - definition (2024), `https://www.ibm.com/topics/threat-actor`, accessed: 2024-04-19
9. Li, F., Zhang, L., Chen, D.: Vulnerability mining of Cisco router based on fuzzing. In: The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014). pp. 649–653. IEEE (2014)
10. Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J.: Fuzzing: State of the art. IEEE Transactions on Reliability **67**(3), 1199–1218 (2018)
11. Mantu, R.A., Chiroiu, M., Tăpus, N.: Network fuzzing: State of the art. In: 2023 24th International Conference on Control Systems and Computer Science (CSCS). pp. 136–143. IEEE (2023)
12. Miller, B.T., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the ACM **33**(12), 32–44 (1990)
13. Munea, T.L., Lim, H., Shon, T.: Network protocol fuzz testing for information systems and applications: a survey and taxonomy. Multimedia tools and applications **75**, 14745–14757 (2016)
14. NC1: Howto: Resizing root partition on x86 – march 2023 edition (Mar 2023), `https://forum.openwrt.org/t/howto-resizing-root-partition-on-x86-march-2023-edition/153398`, accessed: 2024-04-15
15. NIST: Vulnerability - definition, `https://csrc.nist.gov/glossary/term/vulnerability`, accessed: 2024-05-30
16. Notebookcheck: Razer blade 15 i7-8750h gtx 1060 max-q fhd laptop review. `https://www.notebookcheck.net/Razer-Blade-15-i7-8750H-GTX-1060-Max-Q-FHD-Laptop-Review.314146.0.html` (2018), accessed: 2024-04-15
17. Pereyda, J.: boofuzz: Network protocol fuzzing for humans. `https://boofuzz.readthedocs.io/en/stable/` (2023), accessed: 2024-04-11
18. Sinkmanu: Fuzzing SSH key exchange begins. `https://gist.github.com/Sinkmanu/b9c5e633157538e1425e8b739a10bd4c` (2021), accessed: 2024-05-08
19. Taber, S., Schanes, C., Hlauschek, C., Fankhauser, F., Grechenig, T.: Automated security test approach for SIP-based VoIP softphones. In: 2010 Second International Conference on Advances in System Testing and Validation Lifecycle. pp. 114–119. IEEE (2010)
20. Vijtiuk, J., Perkov, L., Krog, A.: Bug detection in embedded environments by fuzzing and symbolic execution. In: 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO). pp. 1218–1223. IEEE (2020)
21. Wang, Z., Zhang, Y., Liu, Q.: A research on vulnerability discovering for router protocols based on fuzzing. In: 7th International Conference on Communications and Networking in China. pp. 245–250. IEEE (2012)
22. Wen, C., Liu, Y., Li, S.: A routing protocols fuzzing method based on man-in-the-middle. In: 2022 2nd International Conference on Frontiers of Electronics, Information and Computation Technologies (ICFEICT). pp. 491–496. IEEE (2022)
23. Yu, B., Wang, P., Yue, T., Tang, Y.: Poster: Fuzzing IoT firmware via multi-stage message generation. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. pp. 2525–2527 (2019)
24. Zscaler: Threatlabz 2023 enterprise IoT and OT threat report (2023), `https://info.zscaler.com/resources-industry-reports-threatlabz-2023-enterprise-ioT-ot-threat-report`, accessed: 2024-05-06