applied sciences

MDPI

Article

EWVHunter: Grey-Box Fuzzing with Knowledge Guide on Embedded Web Front-Ends

Enze Wang, Baosheng Wang, Wei Xie *, Zhenhua Wang, Zhenhao Luo and Tai Yue

College of Computer, National University of Defense Technology, Changsha 410073, China; wangenze18@nudt.edu.cn (E.W.); bswang@nudt.edu.cn (B.W.); wangzhenhua19@nudt.edu.cn (Z.W.); luozhenhao17@nudt.edu.cn (Z.L.); yuetai17@nudt.edu.cn (T.Y.)

* Correspondence: xiewei@nudt.edu.cn

Received: 10 April 2020; Accepted: 8 June 2020; Published: 10 June 2020

Abstract: At present, embedded devices have become a part of people's lives, so detecting security vulnerabilities contained in devices becomes imperative. There are three challenges in detecting embedded device vulnerabilities: (1) Most network protocols are stateful; (2) the communication between the web front-end and the device is encrypted or encoded; and (3) the conditional constraints of programs in the device reduce the depth and breadth of fuzz testing. To address these challenges, we propose a new type of gray-box fuzz testing framework in this paper, called EWVHunter, which is mainly used to find authentication bypass and command injection vulnerabilities in embedded devices. The key idea in this paper is based on the observation that most embedded devices are controlled through the web front-end. Such embedded devices often contain rich information in the communication protocol between the web front-end and device. Therefore, by filling data at the input source on the web front-end and reusing web front-end program logic, we can effectively solve the impact of the stateful network protocol and communication data encryption on fuzzing without relying on any knowledge about the communication protocol. Additionally, we use firmware information extraction to enhance EWVHunter so that it can detect vulnerabilities in deep layer codes and hidden interfaces. In our research, we implemented EWVHunter and evaluated 8 real-world embedded devices, and our approach identified 12 vulnerabilities (including 7 zero-days), which affect a total of 31,996 online devices.

Keywords: gray-box fuzzing; knowledgebase; authentication bypass; command injection; embedded web front-end; stateful network protocol; communication data encryption

1. Introduction

Embedded devices are widely used in our lives, such as smart home routers and IP cameras. In the past few years, the connection between embedded devices has become closer, forming the Internet of Things (IoT). According to a recent report [1], the number of IoT devices has reached 14.2 billion in 2019 and is expected to reach 25 billion by 2021. However, some current research shows that the loose protection of these devices and the security vulnerabilities are widespread and easy to exploit [2–4]. These security risks can allow attackers to build large-scale botnets easily [5] and provide the basis for DDoS attacks. According to Akamai's statistics [6], all 300 Gbps attacks that occurred in 2016 were entirely or partially based on online IoT devices (e.g., IP cameras and home routers). In the fourth quarter of 2016, more than half of the attack traffic came from the Mirai botnet. Therefore, the security of embedded devices is not only related to user safety but also public safety. Thus it is crucial to discover security vulnerabilities in embedded devices in the real world.

The firmware-based analysis is a straightforward approach for finding security vulnerabilities with embedded devices. There may have misconfigurations or security vulnerabilities in firmware [7],

like traditional software. Security analysis on firmware generally contains static analysis on the source code and dynamic analysis on the binary program [8]. Static analysis usually first obtains the firmware image from public channels, and then uses unpacking tools (such as Binwalk, FRAK, and BAT (Binary Analysis Toolkit) [9–11]) to unpack the firmware. After that, we can use static analysis tools to analyze it. Firmware emulation is the base of the dynamic analysis method. If we can emulate it successfully, dynamic analysis tools are used to discover security vulnerabilities. Nevertheless, there are two challenges to a completely firmware-based approach to security analysis. The first is that static analysis is challenging because it targets a specific problem domain (Lua, PHP, binary CGIs, and so forth). However, the embedded device typically contains an amalgamation of various programs and binaries and there is no static analysis tool for that case. The second is that dynamic analysis techniques rely on the emulation of certain architectures [12], while embedded firmware requires specific hardware support. In response to this situation, some studies like Avatar [13] attempt to forward hardware I/O to real hardware so as to improve the firmware's emulation, but this method is less scalable. In this paper, we used fuzzing techniques to discover security vulnerabilities in embedded device firmware. As it can target on a real device, it does not care whether the emulation of the target device firmware can be completed.

Due to the limited hardware resources of an embedded device, it does not have a user interface (e.g., keyboard, screen) like a desktop computer. However, it provides some interfaces for configuration and maintenance. Some early devices mainly rely on custom protocols or traditional interfaces (such as telnet). Nevertheless, with the widespread existence of embedded devices in people's lives, to facilitate operation, a management terminal with a web front-end has become a universal interface. The web front-end packs user inputs into a message according to protocol specifications and sends the message to the device for processing. Therefore, the web front-end contains a lot of information, such as control messages, URLs (Uniform Resource Locator), implementation of stateful network protocol, and the communication message encryption function. The communication protocol between the web front-end and the device can be a standard protocol or a vendor's self-design protocol. When a device accepts a message that conforms to the protocol specification, it processes the message. However, if there are implementation flaws in this process, the device will be vulnerable. Furthermore, there are many embedded devices with a web front-end (e.g., 83.6% in routers and 93.2% in cameras) according to the statistics we showed in Section 3.3. Hence, potential vulnerabilities can be discovered in embedded devices based on web front-end.

Based on the above observations, this paper proposes a framework, named EWVHunter, with a knowledgebase guide for gray-box fuzzing, which is specifically used to detect security vulnerabilities of embedded devices. As our firmware analysis only extracts keywords related to logical judgment and web file paths from the firmware to build the knowledgebase, it does not require a full static analysis of the code in the firmware. The knowledgebase guides gray-box fuzzing based on web front-end. EWVHunter has two unique attributes. The first one is a knowledgebase built on the information extracted from the firmware, which enables us to discover vulnerabilities hidden in deep logic and also allows us to find the hidden developer interfaces without authentication. The second one is that it performs fuzzing based on the web front-end. This way of filling data on the web front-end input sources will directly reuse the original functions to produce meaningful test cases for fuzzing the target firmware. Therefore, there is no need to perform reverse analysis on the communication protocol, which can well support stateful network protocol fuzz testing. Compared with previous work [12], we can also effectively support the case of encrypted data transmission.

To verify and evaluate this knowledge-guided gray-box fuzzing, we designed and implemented a fully functional prototype of EWVHunter and deployed it in an experimental environment to find authentication bypass and command injection vulnerabilities in embedded devices. EWVHunter was tested on eight real-world devices and successfully found 12 vulnerabilities, seven of which were previously unknown high-risk vulnerabilities. We reported these vulnerabilities to corresponding vendors and some of them have released feature updates. We also compared EWVHunter with

WMIFuzzer, AFL, and boofuzz [14–16], and the result showed that EWVHunter performed better than them in ability, efficiency, and effectiveness.

In short, we summarize the contributions of the paper as follows:

- We proposed the first gray-box fuzzing framework based on the web front-end in embedded devices, EWVHunter, for a security analysis of embedded devices. By using the information in the official firmware and the program logic of the web front-end used as the control terminal, EWVHunter can automatically detect authentication bypass vulnerabilities and command injection vulnerabilities in embedded devices;
- We solved three challenges of fuzzing tests towards embedded devices: (1) The difficulty of testing deep layer codes and finding hidden interfaces; (2) the challenge of dealing with stateful network protocols; and (3) the problem of handling encrypted or encoded messages;
- We implemented a prototype of EWVHunter, conducted gray-box fuzzing tests on eight real-world devices and found 12 vulnerabilities, including seven unreported zero-day vulnerabilities, which affect a total of 31,996 online devices.

The rest of the paper is structured as follows. In Section 2, we review the related work. Section 3 gives the background knowledge on the vulnerability discovery of embedded devices. Section 4 discusses a simple example to highlight challenges. Section 5 introduces the methods we have proposed to meet the challenges. We introduce the implementation of EWVHunter in Section 6. Section 7 describes the evaluation of EWVHunter. Section 8 discusses limitations and some possible improvements. Finally, Section 9 concludes this paper.

2. Related Work

The current research work focuses on the analysis of embedded device firmware. There are two methods of firmware analysis: Static analysis and dynamic analysis. Costin et al. [7] used static analysis tools (e.g., BinWalk, FRAK, and BAT [9–11]) to unpack the firmware and then performed large-scale firmware analysis. The unknown vulnerabilities discovered by the author mainly involved hard-coded backdoors, private SSL (Secure Sockets Layer) keys, XSS (Cross-site Scripting), outdated and vulnerable software packages used in firmware, building images as root users, and the webserver configuration that makes many firmware images vulnerable. Firmalice [17] defined the three cases of authentication bypass vulnerability, hard-coded logical judgments in firmware, some hidden interfaces, missing authentication interfaces, and certain vulnerabilities (i.e., remote command execution vulnerability) hidden in the interface. It proposed a binary analysis scheme based on symbolic execution engines and program slicing, which aimed to detect this vulnerability in embedded device firmware. Static analysis of firmware can automatically discover command injection [18,19], but may not find all the vulnerabilities. Additionally, alerting on non-vulnerabilities is also a well-known limitation for static analysis techniques.

Other researches have used dynamic techniques. Costin et al. [20] implemented a fully automated framework based on dynamic firmware analysis technology to find vulnerabilities in embedded web applications. Firstly, they unpacked the firmware and then used RIPS [21] to analyze these situations during the static analysis phase. Secondly, in the dynamic analysis phase, the vulnerability scanning technology is used to verify vulnerabilities. Although the author used some heuristics to locate the web service configuration files and *chroot* to the unpacked firmware for starting the *init* program, this method does not support nonvolatile memories (NVRAM) related operations. Moreover, because of the diversity of web services, the emulation is likely to fail. To increase the success rate of firmware emulation, Jonas et al. [13] proposed another dynamic analysis framework for the security analysis of the embedded device firmware called Avatar. It uses S2E [22] to perform selective symbolic execution in the emulator and forwards I/O operations from the emulator to real devices connected to the framework. In short, there are certain limitations to discovering vulnerabilities using full static analysis or through full firmware emulation. Thus, we used fuzzing technology to discover vulnerabilities in embedded devices without the dynamic and full static analysis.

As most network-capable devices communicate with external entities, some work have proposed fuzzing communication protocols to find vulnerabilities. Fuzzing network protocols is not a new issue. Some early works such as SPIKE [23] and PROTOS [24] are used explicitly for fuzzing network protocols. WSFuzzer [25] and Wfuzz [26] are fuzzing tools for web applications. However, they cannot support the fuzzing of stateful network protocols. They need to at least add state update mechanisms to support it, but this is a challenge. Banks et al. [27] later implemented the SNOOZE fuzzing tool for stateful network protocols. This method allows testers to describe the stateful operations of the protocol and messages that need to be generated in each state. Still, this tool only targets SIP-based network applications. Yu et al. [28] implemented IoTHunter via using a multi-level message generation mechanism to complete the fuzz testing of stateful network protocols. Nevertheless, their method only targets some standard protocols, *snmp, ftp, ssl, bgp, smb*, in IoT firmware. Besides, the multi-level message generation mechanism is no way to apply to situations such as the state transition. Ma R. et al. [29] defined the state of the agreement as to the complexity of communication, relevancy in context, good transaction semantics, and state transition. Furthermore, many protocols also add a transmission message encryption mechanism for security. None of the above tools can handle the case of encrypted transmission messages, although Tsankov et al. [30] implemented a lightweight and modular fuzzy SECFUZZ. It fuzzes against stateful network protocols and can process encrypted messages. However, for the processing of encrypted messages, it is necessary to provide the encryption algorithm and encryption key to SECFUZZ. Moreover, it cannot handle unknown encryption algorithms.

In a recent study, Chen et al. [31] observed that many IoT devices are controlled through mobile applications, so IoTFUZZER was designed. IoTFUZZER is an automatic fuzzing framework based on Android apps for mining memory leaks in IoT devices. It uses taint-based propagation to track atomic data used to construct network messages dynamically and then mutates the atomic data. It uses a mutation-based fuzzing method, so there is no need for a predefined data model. However, not all IoT devices have an Android application client and IoTFUZZER can only detect memory corruption. The authors of WMIFuzzer firstly addressed the issue of fuzzing IoT devices with a highly structured communication message. However, they also mentioned the difficulties in fuzzing stateful network protocol and communication data encryption, which is focused on in this paper. WMIFuzzer [14] is a tool for fuzzing the web management interface of COTS IoT devices. It uses a brute force UI automation to drive the web interface to generate the initial seed message automatically and proposes a weighted message parsing tree (WMPT) to guide the mutation to generate structurally-valid messages. However, it cannot fuzz stateful network protocols (e.g., context-sensitive protection of CSRF tokens [32]) and a transmission message with encryption. Besides, brute-force UI automation will miss certain context-sensitive request events. EWVHunter fills data (with details in Section 5.3) at the web front-end input sources and then reuses program processing logic to construct the structured communication messages. The method could solve the problem of fuzzing to stateful network protocols and encrypted messages. Moreover, our activity path graph could handle context-sensitive request events. AFL [15] and boofuzz [16] are currently the most advanced fuzzers and widely used to find vulnerabilities in software. However, if they are used to find the vulnerability of embedded devices by fuzzing, we need to modify them.

In the process of fuzzing, most of the existing solutions will generate a large number of invalid test cases. That makes it difficult to efficiently penetrate the program logic, so it is a challenge to discover deep bugs by fuzzing embedded device firmware. In the fuzz testing of binaries, Stephens et al. [33] implemented a hybrid vulnerability excavation tool driller. It uses concolic execution to perform inputs that generate complex checks that separate compartments and then fuzzes programs in the compartments to discover vulnerabilities hidden in deep program logic. Yue et al. [34] present a knowledge-learn evolutionary fuzzer named LearnAFL, which is based on AFL. It can learn the partial format knowledge of some paths by analyzing the test cases that exercise the paths and then guide the AFL mutation process according to the format generation theory. This method can effectively explore deeper paths and reduce test cases that use high-frequency paths. However, in fuzzing network

protocols for embedded devices, there is no way to generate test cases for exploring deeper paths. Furthermore, embedded device firmware usually depends on various technologies, which makes it more challenging to calculate logic conditions with a general-purpose concolic execution and use static analysis tools designed for specific techniques to analyze embedded device firmware. We extract the information from the firmware to use to fill input sources, thereby enhancing the breadth and depth in fuzz testing.

3. Background

In this section, we briefly introduce typical smart home environment architecture in Section 3.1 and summarize the difficulties in firmware analysis in Section 3.2. Then, we show the embedded web program structure and explain the important role of the embedded web front-end in communicating with the device in Section 3.3.

3.1. Typical Smart Home Environment Architecture

In a typical smart home ecosystem, as shown in Figure 1, a variety of embedded devices may be deployed depending on the different purposes. Embedded devices can be divided into two types according to nodes: Gateway nodes and sensor nodes [14]. To facilitate user operation, embedded device vendors tend to provide the corresponding mobile applications or web applications, as the control node. The function of the sensor is to collect information and push it to the user or receive instructions from the user and execute them. The gateway node is generally a home router, which is mainly used to connect the embedded device into a network environment, which is convenient for the control node to complete the control of the sensor node through the network node. Thus, the gateway node is a key node that separates the smart home from the public Internet. In addition, the request sent by the control node can modify the system running status. That is why the research cases in this paper mainly focus on control nodes and gateway nodes.

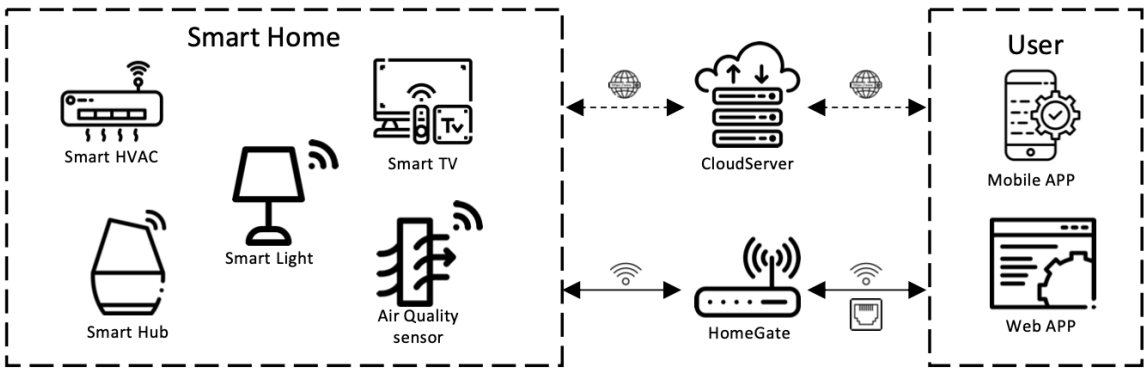

Figure 1. Smarthome communication architecture.

Some embedded devices generally obtain many sensitive data (e.g., sleep patterns, health information, and children's privacy) through sensors. As this information is usually shared with other devices through communication methods such as network and Bluetooth, the security and privacy of data transmission is critical. Due to limitations of hardware resources, only lightweight security strategies can be used on embedded devices. Although there have some researches on lightweight security authentication protocols for embedded devices [35–37], it does not entirely improve the security of the devices.

3.2. Difficulties in Firmware Analysis

There are two main obstacles of firmware analysis in the current security research for embedded devices:

1. It is difficult to perform static analysis on server-side source code from firmware. As embedded web applications usually rely on multiple technologies (such as PHP, binary CGI, CGI composed of bash scripts, Haserl, Lua, and custom server-side languages) and static analysis of source code is generally designed for a specific development language, the existing static analysis tools are difficult to use for achieving good results on the embedded device firmware. In addition, the current static analysis of software technology has a large number of false positives;
2. It is difficult for binary executable analysis on the files from unpacked firmware. Binary executable analysis can be divided into static analysis and dynamic analysis. The hardware architecture of embedded devices is often not uniform (for example, ARM, MIPS, and MIPSel). Therefore, the static binary analysis process for firmware involves a lot of manual and repetitive work. There will be a lot of false negatives (FPs) and false positives (FNs) when only using the static analysis to detect command injection and other vulnerabilities [19]. One method of dynamic binary executable analysis is an emulating running firmware. However, a perfectly running emulation in all the firmware is challenging. For example, the lack of nonvolatile memories (such as NVRAM) is a widespread emulation failure. Three problems will be faced if you only emulate the web service. Firstly, embedded devices often use a modified web server. Secondly, server-side languages are also often associated with binary CGI. Thirdly, it usually uses commands to perform functions such as system to call binary files in the firmware. These situations account for at least 57% of embedded devices [20]. Therefore, the existence of these problems makes it difficult to develop an embedded web application emulation environment suitable for all situations, and dynamic security analysis based on emulation runs is often time-consuming.

3.3. Embedded Web Front-End

As described previously, many vendors usually provide a management terminal with a web front-end because it is lightweight, simple, and easy to implement. Nevertheless, achieving a secure web application is a difficult task. The Gartner Group showed that more than 70% of vulnerabilities are hosted in the (web) application layer, not the network or system layer [38]. In this section, we introduce the structure of embedded web applications and display the popularity of embedded devices with a web front-end through data analysis.

The overall structure of the embedded web program is shown in Figure 2. The web front-end is directly related to the user. It generally composes codes (HTML, JavaScript, CSS) and other static resources and shows the specific UI interface to the user after the browser parses codes. Users can operate on the web front-end to input values or trigger various events (such as clicks and selects), some of which can send network requests. Then, the message is sent to the webserver. The webserver concentrates on parsing the request and sends the result of the parsing to the page handler. The page handler sends the results to the web server after processing, and then the webserver transmits the processing results to the browser in the form of a request-response. The browser parses the request and presents the results to the user. It can be seen that the web front-end mainly focuses on constructing the transmitted messages conforming to the protocol specification in this process. The server behind the device tries to decode or encode that message. The page handler is responsible for processing the decoded message. The design and implementation of this part are diverse, such as PHP, Perl, Lua, and CGI-bin [39–41]. Therefore, if there is a security flaw in the implementation of the webserver or the page handler, the corresponding security flaw may be triggered when the message through the web front-end is transmitted to the device. If we fill data at the web front-end input sources and reuse program logic to construct a message to fuzz the embedded device, it will bypass the analysis of its protocol, thereby making vulnerability discovery easier. Moreover, some previous studies [7,20] have pointed out that many embedded devices have more security flaws. Therefore, it is an excellent way to detect security vulnerabilities in embedded device firmware.

To illustrate that our method is practical, we use the FOFA [42] cyberspace assets search engine for preliminary data analysis. The FOFA can help customers quickly match network assets and speed

up subsequent work, for instance, analysis of vulnerability impact range, application distribution statistics, and application popularity ranking statistics. Firstly, 23 kinds of current embedded web servers were collected. The specific names and the number of devices used on the public network are shown in Figure 3. It can be seen that there are a large number of webservers in embedded devices. After that, we mainly investigated the two types of devices that currently account for the most significant proportion of embedded devices: Routers and cameras. There are 11,482 types of routers and 7899 types of cameras. We investigated these two types of devices, and the detailed data are shown in Tables 1 and 2. We found that the proportion of devices with the web front-end in embedded devices was 83.6% for routers and 93.2% for IP cameras.

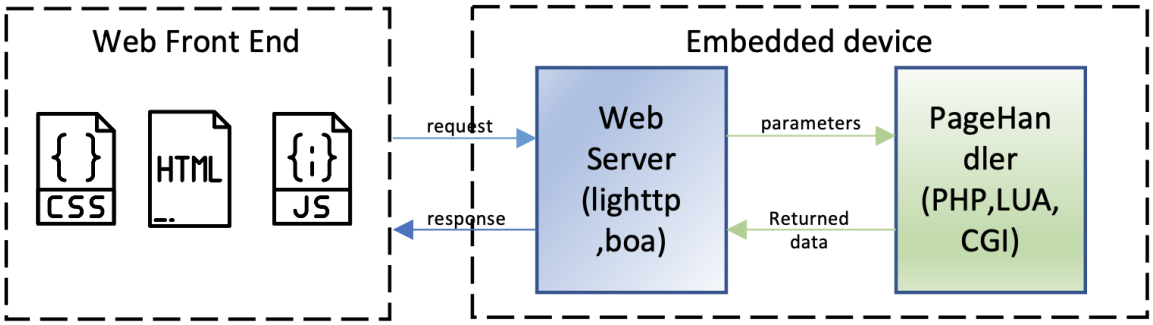

Figure 2. Embedded web application architecture.

Table 1. Online router front-end interface statistics.

Online Devices	HTML	CGI	HTM	ASP	PHP	PERL	JSP	ASPX
27,233,570	22,761,700	2,578,849	1,749,387	947,803	430,264	12,194	7608	2105

Table 2. Online IP camera front-end interface statistics.

Online Devices	HTML	PHP	CGI	JSP	HTM	ASP	PERL	ASPX
330,201	307,940	72,816	40,316	7370	4944	4021	2419	1199

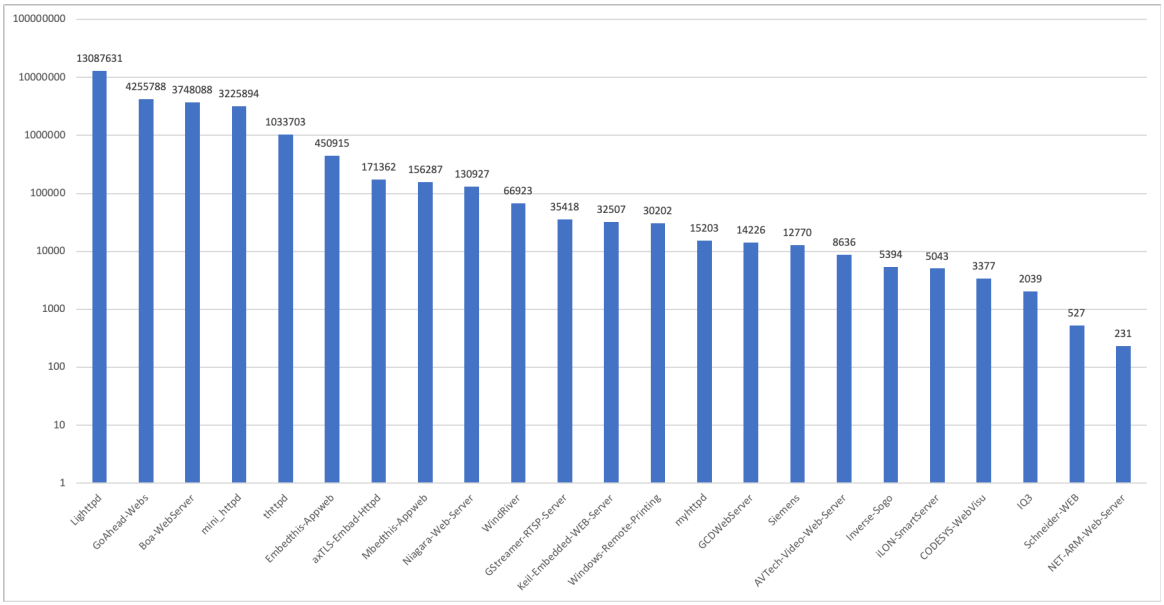

Figure 3. Statistics of online devices using the embedded web server.

Our Methods. From the above situation, it is challenging to analyze the firmware directly to complete the discovery of new vulnerabilities. Thus, we utilize gray-box fuzzing to detect vulnerabilities in embedded devices automatically. After analyzing the existing embedded device communication architecture, it was found that the embedded web front-end played a vital role in interacting with embedded devices. Based on this observation, we discovered that firmware vulnerabilities could be detected in the device based on the embedded web front-end. This solution could avoid reverse analysis of network protocol, reducing the difficulty and complexity of finding new vulnerabilities. It is this discovery that prompted us to design a fuzzing framework based on the embedded web front-end.

4. Challenges

We designed a fuzzing framework that can send probe messages to embedded devices to detect authentication bypass and command injection vulnerabilities. There is currently no unified communications standard for embedded devices, so manufacturers tend to customize data formats and protocols to meet the communication needs of clients and devices. Moreover, most application-layer protocols are stateful [29]. The state of the network protocol is reflected in multiple message interactions that occur when a logical function needs to be performed and while a context needs to be created according to its last state. The universal fuzzing method lacks support for state information and generates a lot of invalid test data, so it cannot fuzz the stateful network protocol. In addition to the fact that the network protocol may be stateful, some of the vendors even use non-standard encryption functions to encrypt messages. There is a simple example to illustrate the challenges that inspire us to propose solutions.

A Simple Example. To better illustrate the impact of stateful network protocol and transmission message encryption on the fuzz testing, we provide a simple program example. The codes in Figure 4 is used as the web front-end and users can control the embedded device through it. There will be a selection box on the web front-end after the page loads and the corresponding input box will appear after selecting an option. After entering characters in the input boxes and clicking the submit button, it will send a HTTP (HypeText Transfer Protocol) request to the server.php. From the data packet in Figure 5, we can see that the input field is encrypted and it has a token. As there is encryption processing, we need to analyze the encryption function when fuzzing this target parameter, which is a difficult point. Furthermore, the message sent carries a token and the data packet will be invalid if the token is modified or expired. Thus, it is a context-sensitive stateful network protocol. This situation invalidates the existing fuzz testing tools.

From the code in Figure 6, we can see that there has a hidden interface "ping_dev" and it is common in real embedded devices. Although developers usually keep the interface for testing purposes, the interface is hidden in the page handle codes and is not displayed in the web front-end. The "ping" interface in this example has a sanitization function (*escapeshellcmd()*), so this is a secure interface. However, the hidden "ping_dev" interface does not have a sanitization function, so it has a command injection vulnerability.

To achieve detecting the command injection vulnerability by fuzzing in the above case, we must first find the hidden interface ("ping_dev") in the page handle code. After that, we can fill the mutation value (e.g., "127.0.0.1 | ls") at the input sources (e.g., "type" and "target") and trigger the web front-end events (i.e., "click") to reuse the program logic related to encryption and protocol realization. Then, we can judge whether the device has a command injection vulnerability according to the page echo result. The token has a timestamp property, so the network protocol is stateful in this example. The method to intercept and modify the packet fields is easy to generate a lot of useless test cases after the timestamp expires and is unable to handle transmission message encryption, so the method can not be used to fuzz this case.

```
<script type="text/javascript">
    function submitForm(){
     if (document.forms[0].target.value !='') {
        if (parent!=null){
            var data_length = document.forms[0].target.value.length.toString();
            document.forms[0].target.value = getEncrypt(document.forms[0].target.value);
  ......
    }
  ......
</script>
  ......
<form action="server.php" name="ping" method="post" style="display:none" onsubmit="return submitForm()" >
    <input type="text" name="type" value="ping" style="display:none;" />
    command:<input type="text" name="target" />
    <input type="submit" value="submit" />
    <input type='hidden' name='user_token' value="<?php echo generateToken(); ?>"/>
</form>
```

Figure 4. Example code from client.php.

Figure 5. Example of web front-end and data packet.

```
checkToken($_REQUEST['user_token']);
 if  ( isset ($_REQUEST['type'])) {
     if  ($_REQUEST['type'] == "ping") {
         if  ( isset ($_REQUEST['target'])) {
             $cmd_target= decrypt($_REQUEST['target']);
             $cmd_target = escapeshellcmd($cmd_target);
             $cmd = 'ping −c 1 ' .  $cmd_target;
             $result  = shell_exec ($cmd);
             var_dump($result);
         }
  ......
     }  elseif ($_REQUEST['type'] == "ping_dev"){
             if  ( isset ($_REQUEST['target'])) {
             $cmd_target= decrypt($_REQUEST['target']);
             $cmd = 'ping −c 1 ' .  $cmd_target;
             $result  = shell_exec ($cmd);
             var_dump($result);
```

Figure 6. Example code from server.php.

Challenges. We have abstracted the challenges in detecting embedded device vulnerability by fuzzing. If we want to generate messages for fuzzing without knowing the state of the protocol and the message encryption, and to also extend the depth and breadth of the fuzzing, we need to address the following challenges.

- ***Challenge 1: Program logic constraints restrict the depth and breadth of fuzzing.*** The message transmitted from the web front-end to the device usually contains multiple parameters. The values of some parameters generally correspond to the logical judgment conditions of the program. If the specific program logic cannot be met, the payload is difficult in entering the vulnerable code

block or find the hidden interface. Therefore, we need to analyze the firmware and extract the value that relates to the device program's (hidden) logical judgment conditions;

- ***Challenge 2: Stateful network protocol.*** A fuzzer is intelligent and often depends on whether it knows the protocol format in advance. It can be regarded as a valid input accepted by the program when the fuzzer generates a message conforming to the protocol format. It can be seen in Section 7.1 that the protocols used by various embedded devices vendors in their products are also different and are not standard protocols. If the device uses a standard protocol, fuzzing may be much more straightforward. The test cases generated by the fuzz testing method without protocol status information is easily rejected by the program, so fuzzing stateful network protocols is a difficult problem. Therefore, we need to adopt some ways to generate valid test cases (particularly messages) to fuzz the device;
- ***Challenge 3: Encrypted or encoded messages.*** Our framework must encrypt or encode the data sent in the same way if the communication between the embedded web front-end and the embedded device is encrypted or encoded. If the target adopts standard encryption or encoding methods, we can also identify the function and then extract the information from the embedded web front-end to re-encrypt or re-encode. However, if non-standard decryption or encoding methods are used, we need to analyze the encrypt functions. Nowadays, to prevent the program from being cracked, the code will be obfuscated in most cases, which makes reverse analysis very difficult. The program will not process the data we transmit if the data cannot be appropriately encoded or encrypted. Therefore, we need some methods to reuse the encryption or encoding functions in the embedded web front-end;
- ***Challenge 4: Monitor the triggering of vulnerabilities.*** Our primary concern is the command injection vulnerability and the authentication bypass vulnerability. For the command injection vulnerability detection, if the HTTP response message does not contain the result returned after the command is executed, we have no way to know when, in real-time, the vulnerability is found and which payload is used. For the authentication bypass vulnerability detection, it is necessary to detect changes in the response content remotely. Therefore, we need to design an effective strategy to detect vulnerabilities in this situation.

Solutions. Fortunately, the following methods have been devised to address the above challenges.

- ***Build a knowledgebase to guide gray-box fuzzing.*** The knowledgebase contains five parts. The website map contains the target URLs for fuzzing. The activity path graph consists of web front-end events and input points. The key information dictionary is used to extend the depth and breadth of fuzzing by satisfying the program logic condition. The mutation strings generated from strings in the payload dictionary are used to probe vulnerabilities. The web front-end analysis phase will use the heuristic dictionary to fill the input sources, then trigger events one by one and record the activity path that can trigger the network request;
- ***Filling data at the web front-end input sources.*** The reverse engineering of stateful network protocols and transmission message encryption is very expensive, so we propose a fuzz testing method that uses data (the strings in the key information dictionary or the mutated strings generated by the strings in the payload dictionary) to fill the web front-end input sources. Then, the original program logic in the web front-end is reused to generate transmission messages that conform to the protocol specifications. Therefore, we do not need to re-implement functions related to communication message encryption and stateful network protocol;
- ***Vulnerability detection strategy.*** As the command injection and authentication bypass vulnerabilities will not cause the device to become unresponsive, we design correspondingly remote vulnerability detection strategies.

5. EWVHunter

In this section, detailed solutions to address the challenges of detecting vulnerabilities in embedded devices via gray-box fuzzing are described.

5.1. Firmware Analysis to Obtain Vital Information and Site Map Construction

We mainly get two kinds of information from the firmware to build the original knowledgebase, one being the strings used for program logic judgment, and the other being the information about the website path.

We extract strings used for condition judgment and hidden in the firmware. If only the web front-end information is used, it is easy to miss the logical entry point of some key code block programs [43]. Thus we propose a static search solution for the firmware. This solution uses regular expressions to match the strings (e.g., hidden options) from the contents, which we get by using the "strings" command to handle the file of the unpacked firmware. We make sanitization rules to deal with these strings. For example, we will replace the value after the equal sign for the strings "AUTHORIZED_GROUP=%d". Then, we use sanitization strings to build a key information dictionary and add the dictionary to the knowledgebase. Later, during the fuzzing process, we use the strings in the key information dictionary to fill the web front-end input sources. The fuzzer can cover the program logic possibly and goes deep inside the program to find the vulnerability.

The web file path information in firmware is mainly used to build an optimized site map. We use a site map to guide the embedded web front-end analysis by telling our fuzzer which URL should be analyzed. Ordinary web vulnerability detection programs only crawl websites to build site maps. However, this method is less efficient and often starts from a URL as an entrance. It is easy to miss some interfaces, especially hidden interfaces and thus miss some vulnerabilities [43]. Therefore, we collect the original site map information in the following three ways. Firstly, we determine the corresponding root directory starting point by retrieving crucial file directories (*/bin/, /sbin/, /HANP/, or /www/*) or crucial files (*/init, /linuxrc, or /bin/sh*). Then, we use the traversal file path method to extract the file (related to the web application) path information. Secondly, we utilize the "strings" command to get the strings in the file of the unpacked firmware, and then apply regular expressions to match the web page routing information from the strings. Thirdly, we obtain the site path from the web crawler. Finally, we combine the three pieces of information and delete duplicate data to get the original site map information. However, the web file path information of the original site map information is not all the web routing information. It may be an absolute path relative to the root file system. Therefore, the path information is combined with the HTTP response status code detection to construct the optimized site map and will be used as part of the knowledgebase.

The site map constructed in this way may contain some hidden interfaces or some test pages left by the developer. These pages are most likely to have security vulnerabilities. Therefore, we use the optimized site map to guide the fuzzing that has the following two advantages. Firstly, the optimized site map can make the fuzzing performed efficiently because the fuzzer does not waste time analyzing unnecessary pages, such as CSS, JavaScript, and image files [43]. Secondly, it increases the chance of fuzzing the interfaces hidden in the firmware.

We are targeting firmware that can be successfully decompressed. The decompression methods for encrypted firmware are temporarily outside the scope of this article. However, for most encrypted firmware we can also use specific techniques to decrypt it and decompress it. These methods can be implemented as EWVHunter extensions in the firmware analysis phase.

5.2. Web Front-End Analysis and Activity Path Graph Construction

Since our goal is to discover security flaws in embedded devices, we only focus on those events related to network requests in the web front-end. Therefore, the purpose of web front-end analysis is to determine which trigger path for the web front-end event can send an HTTP request when the last event on the path is triggered. Besides, we develop simple triggering strategies to build the activity path graph to promote efficiency in the fuzzing phase.

The web page loading takes time and this time may be related to many factors. If we cannot accurately grasp the loading status of the page for analysis, the information analyzed is inaccurate or invalid. Furthermore, network-related resources can be loaded for a long time or cannot be loaded

due to network congestion. If the fuzzer waits for such resources to load, it will seriously affect the efficiency of fuzz testing.

To improve the fuzzing efficiency, we determine whether the loading state of the page meets the requirements for starting the web front-end analysis using the algorithm as shown in Algorithm 1. There are three web front-end events directly related to page loading, namely load, DOMContentLoaded and networkAlmostIdle. The load event is fired when the whole page has loaded, including all dependent resources. The DOMContentLoaded event is used to indicate that the initial HTML document has been fully loaded and parsed, but it is not sure whether the style sheet, image, and child frame have finished loading. The networkAlmostIdle event is used to determine if any network requests have not been completed. Before the browser fully displays the web page, it needs to make a series of network requests with the server to get all the resources presented by the web front-end. After understanding the role of these three events and analyzing the page loading situation in various network situations, we set the following priorities for these three events, and judge the page loading situation based on ensuring efficiency and accuracy. We set the load event priority here, the DOMContentLoaded event second, and the networkAlmostIdle last. In particular, firstly we set a flag F to indicate whether web front-end analysis can begin. Then we send a network request. Finally, we monitor the loading of web resources, set the time to wait for each event to complete and determine the completion of the event. If the value of F is set to 1, we end the observer of the web front-end loading event and start the web front-end analysis.

Algorithm 1: Page load state detection algorithm

Input: u: URL of the target device
Output: F: The flag determining whether page analysis is possible

```
initial F = 0;
send_requests(u) ;  // waiting for the page load event according to the following strategy after sending the request
if The "load" event completed then
    F = 1 ;                          // The F value shows that we can start web front-end analysis
else if Wait for the "load" event to timeout && The "DOMContentLoaded" event completed then
    F = 1;
else if Wait for the "DOMContentLoaded" event to timeout && The "networkAlmostIdle" event completed then
    F = 1;
return F;
```

After making sure that the page is fully loaded, we can start the web front-end analysis. In order to fuzz every request from the web front-end to the embedded device entirely and efficiently, we build an activity path graph. We first analyzed the common binding events on the front-end HTML tags (as shown in Table 3). These events are commonly used to trigger page changes or send network requests. These events are also often associated with codes that implement the stateful network protocol or communication encryption. Later, a simple strategy is formulated according to the common execution order of events. The strategy is to traverse (e.g., all options of the select event) and trigger all web front-end events. When our fuzzing framework encounters the input source, it will use the heuristic dictionary in the knowledgebase to fill it. In the process, we will observe changes in the page and the sending of network requests. The events that caused the page change are used as the initial or intermediate state node, and the event is triggered to send the network request as the final state node. During front-end analysis, the value (such as default values and option values) on the HTML tag will be recorded and stored on the path between the nodes. Finally, we build the activity path graph based on this strategy.

Table 3. Events common on HTML tags.

Events	Button	Select	Input	a	Textarea	Span	td	tr	div
click	✓	✓	✓	✓	✓	✓	✓	✓	✓
dblclick	✓	x	x	✓	x	x	x	x	x
change	x	✓	✓	x	✓	x	x	x	x
mouseup	✓	✓	x	✓	✓	✓	✓	✓	✓
mousedown	✓	✓	✓	✓	✓	✓	✓	✓	✓
keyup	✓	x	✓	x	✓	x	x	x	x
keydown	✓	✓	✓	✓	✓	x	x	x	x
blur	x	✓	x	✓	x	x	x	x	x
focus	x	x	✓	x	✓	x	x	x	x
scroll	x	x	x	x	x	x	x	x	✓

For example, as shown in the simple example in Section 3.2, we can construct an activity path graph in Figure 7 with three triggering network request paths. The differences between "echo", "ping", and "ping_dev" options will lead to three different paths. After filling the input source, we can click "submit" to send a network request.

In the fuzzing phase, we use data to fill input sources in the activity path and trigger orderly the event in the activity path.

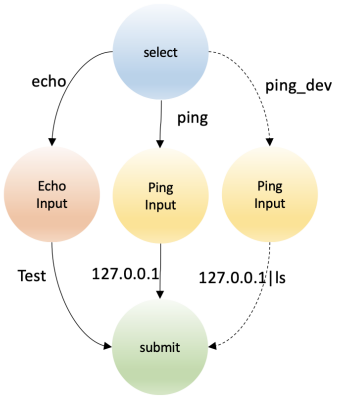

Figure 7. Activity path graph. The circle node represents an event. Inputs between events are marked on the wires of the nodes.

5.3. Trigger the Vulnerability by Fuzz Testing

We fill data into the web front-end input sources at runtime to solve the problem of stateful network protocol and transmission message encryption in fuzzing. The values accepted by the input sources in the embedded web front-end usually consist of hard-coded characters, user input, and return values from system APIs. From the web front-end point of view, these three contents can be modified. The stateful network protocol is a communication process between the client and embedded device. If we use reverse analysis and rewrite the program to achieve this process, it is undoubtedly very challenging. Therefore, we fill the data that need to be used in the protocol message from the input sources. After that, our framework will trigger web front-end events according to the activity path graph and reuse the original program logic in the process. After being processed by the original program logic, these data will become protocol fields and then be sent to the embedded device. We modify the protocol field parameters from the embedded web front-end to generate communication messages, which has two benefits:

1. For stateful network protocols or communication message encryption, the fields can be fuzzed before the program processes them;
2. Without the reverse analysis of the protocol implementation, the fuzzing of unknown protocols can be completed.

Fuzz scheduling. If all protocol fields are mutated at the same time, the constructed message is likely to be a message that does not conform to the device processing logic. As a result, the probability of rejection by embedded devices is high and fuzz testing is invalid. In this case, we designed a fuzz testing scheduling algorithm to assign the web front-end input sources to be filled, as shown in Algorithm 2. For each message, we want to randomly select a subset of the fields for mutation, not all fields. In particular, firstly, we extract the input sources set D from an activity path. Then we take the concept of combinatorial number and use all combinations of the input sources in the web front-end to build a filled position set *positionSet*. As the activity path that can trigger network requests to improve the efficiency of fuzzing will be used, the set *positionSet* is also a set of input points on an activity path. We will then randomly select the elements *positionList* from the set *positionSet*, which identifies which input points will be filled. Finally, the fuzzer can complete the mutation based on the strings in P in Definition 1 and fill this mutation strings into the input sources specified by the set *positionList*. The network request will be sent after the last event on the activity path is triggered.

Definition 1. *The knowledgebase constructed in this paper and used to guide fuzz testing is a 5-tuple KB = (P, H, K, S, M), where:*

1. *P is the payload dictionary used to fuzz the embedded devices in which $p \in P$ is used as initial mutation strings;*
2. *H is the heuristic dictionary used to fill input sources to build the activity path graph;*
3. *K is the key information dictionary extracted from firmware and used to extend the depth and breadth of fuzzy testing by satisfying the program logic condition;*
4. *S is the website map in which $u \in S$ is called the URL;*
5. *M is the set of the activity path graph: $M_i \in M$ in which $g \in M_i$ is called a path on the activity path graph.*

Algorithm 2: EWVHunter Input Sources Random Fuzzing Algorithm

Input: g: a path in the activity path graph
P: set of payloads in the knowledgebase
K: set of key informations in the knowledgebase

$D = \{d_1, d_2, ..., d_n\}$ = get_input_sources_from_path(g) ; // a set of input sources on an activity path graph

positionSet = $\{a_1, a_2, ..., a_m\}$ = get_position_combination_set(D) ; // a set of various combinations of filled positions,(m=2^n)

while *positionSet not Null* **do**

  positionList = $\{pl_1, pl_2, ..., pl_i\}$ = random_get(*positionSet*) ; // *positionList* is an element randomly selected in *positionSet*, and this element will be removed from the set after selection. *positionList* is a combination of filled positions in which $pl_i \in$ *positionList* is called a filled position.

  fill_data_to_input_source(*positionList*, mutation(P), K) ; // The mutation will be based on the strings in P, and the strings will be filled in the position in the *positionList*

  ordered_trigger_event(g) ; // After filling the web front-end input sources, the EWVHunter triggers the event according to the event trigger path indicated by g

Fuzz strategy. There are two strategies for fuzzing, one is generation-based fuzzing and the other is based on mutation. Generated fuzzing needs knowledge of the format of the target to create test cases. Mutation-based fuzzing makes small changes based on the seed with the type information. However, the protocol targeted in this article is highly-structured data. If the traditional method of random byte mutation on a binary file is adopted, numerous invalid test cases will be generated and using fuzzing to detect deeper code layer vulnerabilities is challenging. So the mutation strategy implemented by EWVHunter is to fill the mutation data from input sources based on the knowledgebase. Specifically, the mutation strategy used in this article is as follows:

1. **The input source mutation is based on payloads in the knowledgebase.** This article focuses on command injection vulnerabilities and authentication bypass vulnerabilities. We observed a critical message that the command injection vulnerability mainly relies on injecting a valid payload from a data source. Therefore, this feature can be used to build an active payload dictionary for vulnerability discovery. To this end, we combined the open-source project FuzzDB [44], PayloadsAllTheThings [45], and the command injection vulnerability payload disclosed in the CVE (Common Vulnerabilities and Exposures) vulnerability library to build a payload dictionary. FuzzDB and PayloadsAllTheThings are two open-source projects that integrate payloads for vulnerability discovery and are used in most security tools. The vulnerability payloads in the CVE vulnerability database are a manifestation of the security research experience. It is easy to have similar vulnerabilities for similar input points in embedded devices, so these payloads have an excellent detection effect. EWVHunter mutates each byte in the payload based on this payload dictionary and then uses the mutated data to fill the input sources;
2. **Filling strings from key information dictionary in the knowledgebase into the input source.** In the knowledgebase, we built a key information dictionary extracted from the firmware to fill the input sources. The reason why this dictionary is used to fill the input fields in the web front-end is that we want the seeds to discover the device program hidden logic (the breadth of fuzz testing). Alternatively, that can let the seeds meet more logical judgment condition, and then make it reach the deep code (the depth of fuzz testing) to fuzz;
3. **Payload priority.** After observation, if we can find a command injection vulnerability in the embedded device by using one of the payloads in the knowledgebase, other input points in this embedded device are likely to have a similar vulnerability. Development thinking determines the existence of this phenomenon. In the embedded device, there are many function points for controlling the back-end to execute commands through the front-end, and the code implementation of these functions is similar. Developers are likely to call the same function repeatedly or use the same code in the corresponding place. Therefore, if a command injection vulnerability is detected in an embedded device, and then this attack payload is set to the highest priority to fill and discover other input points. The efficiency of fuzz to find the vulnerability can be improved.

5.4. Vulnerability Detection Strategy

Previous research on fuzzing the protocol was to identify whether the device's program was crashing or not responding to the network when a vulnerability appeared. However, this method cannot detect command injection and authentication bypass vulnerabilities. Besides, embedded devices generally do not support local monitoring, so we cannot obtain much information (for example, file changes in the file system, CPU overhead, running process information). We make strategies through network communication to remotely detect command injection and authentication bypass vulnerabilities. The discovery of command injection vulnerabilities is mainly divided into vulnerability detection with echo results and without echo results after executing the command. The detection of authentication bypass vulnerabilities mainly relies on response status code and response content. In this paper, the vulnerability detection strategy can be divided into the following ways:

1. **The result is displayed after the command is executed.** In this case, the result of the command execution will be displayed on the front page or the response of the request. Then it is easy to detect the corresponding vulnerability at this time;
2. **No result is displayed after executing the command.** In this case, there is no way to determine directly whether the embedded device executed our payload. Thus, we use out-of-band data transmission, time blind injection, file creation, and reboot to detect the target. The way to transfer data out-of-band is to use network operation commands such as "wget", "curl", and "ping" to include special characters in the request, and the OOB (Out-of-band) platform can receive the request. Afterward, we only need to determine whether these messages carry corresponding

special characters to detect whether a vulnerability has occurred. The time blind injection method mainly executes commands that can trigger time delay, such as sleep. It determines whether the attack payload successfully triggered the response vulnerability by detecting whether the response corresponding to the request is delayed by the corresponding time. The way to create a file is to create a file under the website's web path, and then this file may be requested from the web request. The way to restart the embedded device using the reboot command is to verify whether the embedded device is not responding to the request;

3. **Interface response status information.** For the authentication bypass vulnerability, the test object is the interface that handles data and website management. The detection method first detects the response data and status code of the page in the unlogged state. Then, it compares the response data and status code of the page in the logged state with the unlogged state after modifying the data packet. The modification of the data packet way is to remove the login information such as cookies, and then add some payloads to the data packet. Then, we observe whether the modified data packets can take login status.

6. Implementation

Our goal is to build a scalable and efficient knowledge-guided gray-box fuzzer. The fuzzer takes into account the dynamic characteristics of the web front-end, the stateful network protocol between the client and the device, and the encryption and encoding of communication data. Our method is implemented in a fuzzer named EWVHunter, as shown in Figure 8. In order to meet scalability, our method was divided into the following steps: 1. Firmware analysis phase; 2. Website map construction phase; 3. Embedded web front-end analysis phase; 4. Fuzz testing phase; and 5. Vulnerability detection phase. After obtaining the device firmware, the first step was to perform a static analysis of the firmware to build a key information dictionary and extract an original site map information. Then, the site map was constructed from the original site map information extracted from the firmware. In the embedded web front-end analysis phase, the primary purpose was to collect the activity paths that can trigger network events and build an activity path graph. The obtained information was applied to make the knowledgebase afterward. The knowledgebase contains the following contents: Processed keywords, heuristically populated dictionaries, payloads for fuzz testing, optimized site maps, and activity path graph that can trigger network events. To improve the efficiency of fuzzing and capture key HTTP request information, we also hooked vital functions (e.g., alert, setTimeout, and send) during the fuzzing phase. The fuzzer based on the embedded web front-end would use the information in the knowledgebase for fuzzing. We used vulnerability detection strategies to detect vulnerabilities during fuzz testing.

We implemented a full-featured prototype of EWVHunter, totaling approximately 3000 lines of python code and 1000 lines of javascript code. In addition, our fuzzer also integrates multiple open source projects (for example, chromium [46], pyppeteer [47], fuzzdb, PayloadsAllTheThings, and vtest [48]) to avoid reinventing the wheel.

The firmware analysis phase. We implemented it with Python and used the system tools "strings" and unpacking tools to complete the static analysis of the firmware. At this stage, we utilized unpacking tools (Binwalk, FRAK, and BAT) to unpack the firmware and the "strings" command to get the strings in the file of the unpacked firmware. Then, we applied regular expressions to match the possible parameter values associated with program logic conditions and the web page routing information from the strings.

The site map building phase. Our first step was to use the request library in Python to help locate the website routing path for web files (such as cgi, html, css, js, jpg, and other files). The second step used a dynamic crawler to obtain hidden link information in the web page. Then build a site map based on the obtained website path information.

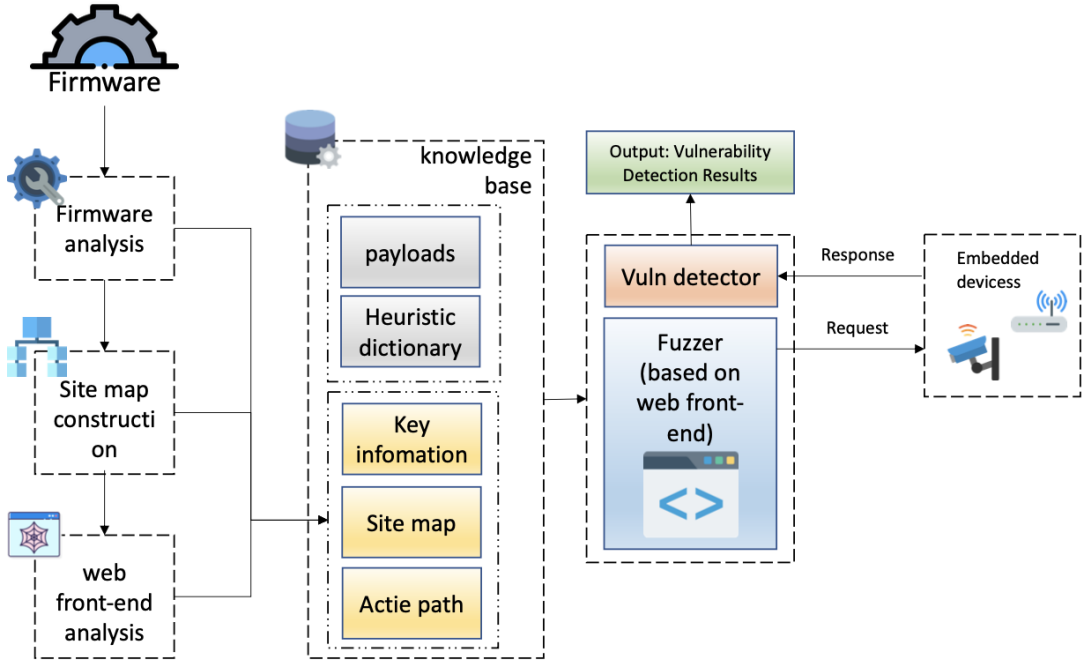

Figure 8. Overview of EWVHunter.

The embedded web front-end analysis phase. Python code was written and pyppeteer was used to communicate with chromium to control the behavior of chromium. We first obtained the web front-end page after successfully loading and then used pyppeteer to control chromium to execute javascript code on the corresponding web page to trigger the web front-end events. A heuristic dictionary was used to populate data when we encountered the data that needed to be populated (e.g., options, user input). In this process, it would detect whether a network request was sent after completing a certain event. If a network request was sent, it would record the corresponding activity path (including the location of filling data and the triggered events). All activity paths of a single page constituted the activity path graph required.

The fuzzing phase. We used pyppeteer to control the chromium to complete the loading of the web front-end, and in combination with the previously built knowledgebase, fill the data and trigger network events from the web front-end according to the activity path graph. Scheduling and strategies for fuzzing were also implemented with Python.

Vulnerability detection phase. Feature strings were added to the payload to increase the accuracy of vulnerability detection. The out-of-band data transmission method to detect vulnerabilities uses the open-source vtest platform. We used a self-built strategy base to detect command injection vulnerabilities and authentication bypass vulnerabilities (with details in Section 5.4).

7. Evaluation

In this section, we evaluate EWVHunter. We first introduced our experiment setup in Section 7.1 and then presented a case study on how EWVHunter found vulnerabilities in Section 7.2. The effectiveness of EWVHunter through the discovery of known vulnerabilities, the discovery of unknown vulnerabilities, and the web front-end analysis module is illustrated in Section 7.3. Finally, the superiority of this study is demonstrated by comparison with similar work in Section 7.4.

7.1. Experiment Setup

Test environment. EWVHunter runs on an Ubuntu 18.04 virtual machine configured with an Intel Core i9 quad-core 2.3 GHz CPU and 8 G RAM.

Dataset selection. We selected eight kinds of real-world embedded devices, mainly router devices that include web front-end. As this article adopts the gray-box fuzzing scheme and needs to extract information from the firmware, the firmware of the selected embedded device can be unpacked using the Binwalk [9] and BAT [11] tools. Details of these devices (type, vendor, model, firmware version, and communication protocol) are described in Table 4.

Table 4. Summary of embedded devices under testing.

Device Type	Vendor	Device Model	Firmware Version	Page Handle	Protocol and Format
Router	D-Link	DIR-859	A3-1.06	php + cgi	HNAP, XML
		DIR-846	100A35	php + cgi	HNAP, JSON
		DIR-412	A1-1.14WW	php + cgi	HNAP, XML
		DAP-1320	A2-V1.21	cgi	HNAP, XML
	Netgear	DGN2200	1.0.0.58	cgi	HTTP, key-value
		DGN1000	1.1.00.48	cgi	HTTP, key-value
	Trendnet	TEW-733GR	v1.03	php + cgi	HTTP, key-value
	Tenda	AC9	2.22.15	cgi	HTTP, key-value

7.2. Real-World Case Study

The DIR-850L Wireless AC1200 Dual-Band Gigabit Cloud Router is a compelling case. Users can access the web front-end for management through a browser. The device's firmware has been released on the official website and can be unpacked using binwalk. Its web front-end consists of HTML and JavaScript, and the web page handle implementation is a combination of modified PHP and CGI. There are currently no tools to analyze the modified web page handle composed of PHP and CGI. So we used EWVHunter to fill data in the web front-end input sources and to perform gray-box fuzzing. The format of the protocol transmission data is XML, and its protocol status is reflected in the relevancy in context.

If we modify the transmission information by capturing and replaying packets, it will be invalid for setting the input point of D-Link's Internet time server in the "tools_time.php" page. Figure 9 shows that the modified value at the server tag in the packet and replay the data.

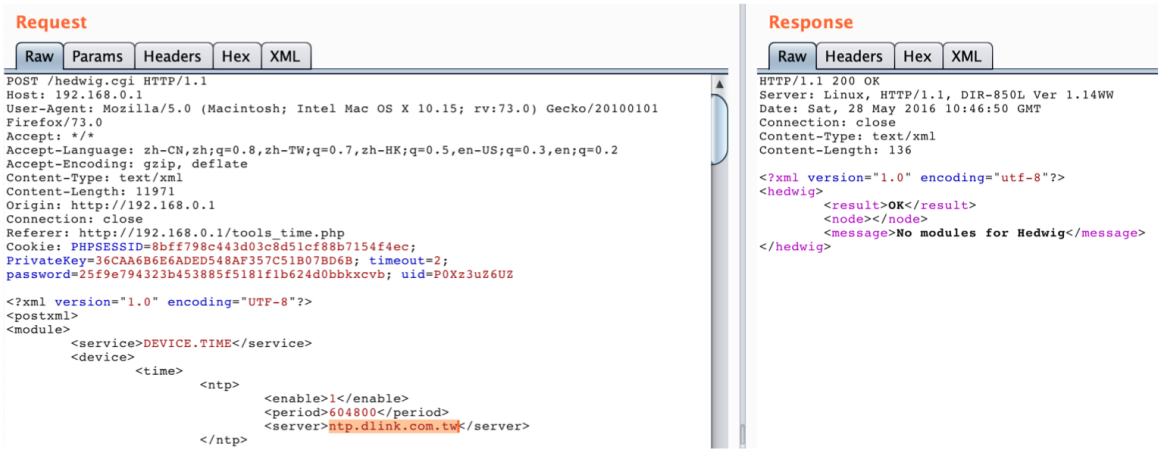

Figure 9. Modify and replay the packet.

Although after replaying the data packet, a response was given with a status code of 200, and resulted in a reasonable response result. However, according to the results displayed on the page, it was still the previous value "ntp1.dlink.com", and it had not changed. Figure 10 shows the web front-end after replaying that data packet.

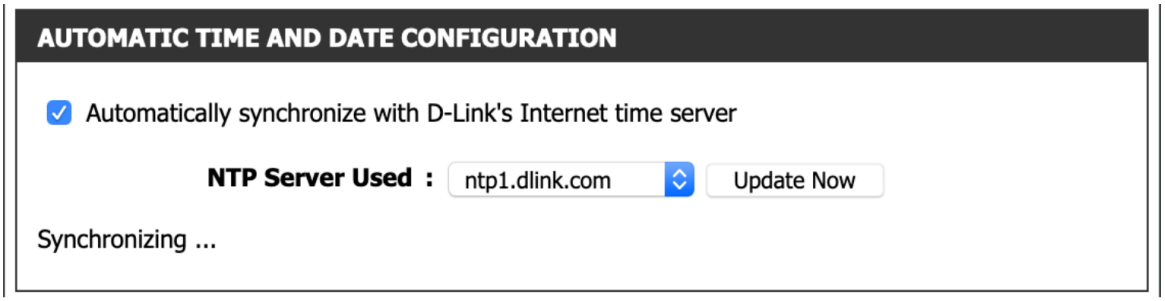

Figure 10. The website page displayed after modifying and replaying the packet.

The reason for the case is that after sending a data packet to the interface "hedwig.cgi", the web front-end sent another data packet to the "service.cgi" interface to complete the setting. Therefore, the state of the protocol is reflected in relevancy in context. If we adopt the method of populating data at the input sources, we can automate this process and let our data be received and processed by the embedded device. Next, we manually demonstrated the process of filling data at the input sources, triggering, and verifying the vulnerability in Figure 11.

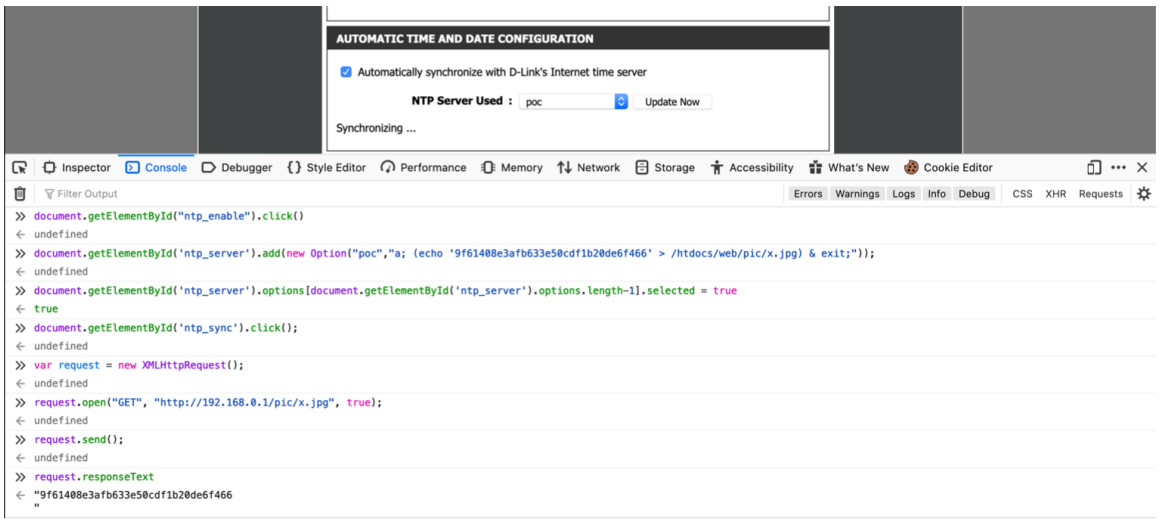

Figure 11. Manual data source mutation, vulnerability triggering, and vulnerability detection in the browser.

The command injection vulnerability was reported after 42 min 34 s. We analyzed the cause of the vulnerability and why EWVHunter could find it. The discovery of the "tools_time.php" page relied on the site map in our knowledgebase. The "Automatically synchronize with D-Link's Internet time server" option on the web front-end was disabled by default. Therefore, if we wanted to automatically trigger the vulnerability, using a violent UI trigger method was not feasible. According to the activity path constructed by EWVHunter, first, the automatically synchronize option needed to be turned on. Then input value of the selection box needs to be set, and this step is to fill mutation data at the input sources. Finally, the "Update Now" button was selected to complete the network request trigger. The mutated data was transmitted to the embedded device after the program processing and received and processed by the device. The payload that could trigger this vulnerability is "a; (echo'9f61408e3afb633e50cdf1b20de6f466 '> /htdocs/web/pic/x.jpg)&exit;". The hash value of the md5 algorithm could enhance the accuracy of vulnerability verification. Moreover, the detection of the command injection vulnerability was parallel, and the hash value could help locate which packet triggered the vulnerability because the sent request was saved in the database. As the function of the payload was to create a file under the web path, the strategy for detecting this vulnerability was to

directly request the corresponding file ("http://192.168.0.1/pic/x.jpg") and verify whether the file content was the hash value in the payload to detect the vulnerability accurately.

After our manual analysis, we found that the page handle file that had processing problems was in file "/etc/services/DEVICE.TIME.php". The "$SERVER" variable was spliced into the string of the command execution, and the developer forgot to sanitize this variable. So, it had a command injection vulnerability, and Figure 12 shows this code snippet. Thus the attacker could inject an evil content code into the string and execute it as additional commands.

```
1  $enable = query("/device/time/ntp/enable");
2  if ($enable=="") $enable = 0;
3  $enablev6 = query("/device/time/ntp6/enable");
4  if ($enablev6=="") $enablev6 = 0;
5  $server = query("/device/time/ntp/server");
6  ...
7  if  ($enable==1 && $enablev6==1)
8  ...
9              'SERVER4='.$server.'\n'.
10 ...
11             '    ntpclient  -h $SERVER4 -i 5 -s -4 > /dev/console\n'.
```

Figure 12. Code from D-Link DIR-850L.

7.3. Effectiveness

We used the automated testing framework EWVHunter (testing each device for 24 h) to fuzz eight real-world embedded devices and found 12 vulnerabilities including seven zero-days. After EWVHunter identified these vulnerabilities, the correctness and harmfulness of the vulnerabilities was further verified.

These command injection and unauthorized access vulnerabilities we found using EWVHunter had a high impact. The command injection could cause these commonly-used embedded devices to be remotely attacked and controlled. Attackers could use these vulnerabilities to execute remote commands, obtain critical information, control devices to build botnets, or use devices as attack springboards to attack other devices on their network. There were mainly information leakage problems for unauthorized access vulnerabilities, such as the leakage of Wi-Fi connection passwords and the leakage of administrator login accounts. Attackers could use this leaked information to connect to the target intranet for further attacks or use the leaked administrator account to log into the target system.

7.3.1. Known Vulnerabilities Verification

To verify that our work is effective, we first use EWVHunter to perform fuzz testing on vulnerable devices. The vulnerable devices were two Netgear routers, one Tenda router and one Trendnet router. These four devices are known to have three command injection vulnerabilities and two authentication bypass vulnerabilities. The vulnerability information to be verified and the devices are shown in Table 5.

There is a simple type of command injection vulnerability in Netgear DGN2200. Its communication protocol is HTTP and the transmission data is the key-value type of POST. The communication messages do not have problems of protocol status and transmission message encryption. The HTTP response data contains the result of command execution, so the strategy for detecting this vulnerability is to verify whether the content in response data is the expected result after command execution. Tenda AC9's message is also transmitted through the HTTP protocol and the transmitted data format is key-value. Its two command injection vulnerabilities are the same type. If we used the fuzzing scheduling strategy in this paper, we could quickly find similar vulnerabilities after finding a vulnerability. Since the command injection vulnerability does not return a result in the HTTP response, we detected whether the payload that has a file creation feature had created a corresponding file (vulnerability detection strategy related to file creation). Both types of vulnerabilities could be detected using EWVHunter.

Table 5. Summary of public vulnerabilities.

Device	Vul-ID	Vulnerability	Remotely Exploitable	CVE-ID
Netgear DGN2200	1	Command injection	True	CVE-2017-6077
Tenda AC9	2	Command injection	True	CVE-2019-5071
	3	Command injection	True	CVE-2019-5072
Netgear DGN1000	4	Authentication bypass	True	N-day
Trendnet TEW-733GR	5	Authentication bypass	True	CVE-2018-7034

The Netgear DGN1000 will skip authentication when the "currentsetting.htm" substring is detected in the URL, and this case is an authentication bypass vulnerability. The authentication bypass vulnerability in Trendnet TEW733GR is that when the transmitted data string contains "AUTHORIZED_GROUP=1", the page considers this to be an authorized user. The two strings "currentsetting.htm" and "AUTHORIZED_GROUP=1" were extracted from the firmware, and we could do this automatically in the firmware analysis phase. Subsequently, this information would be stitched into the request data to probe which pages or interfaces have authentication bypass vulnerabilities. The detection strategy adopted was to check whether response data was the content returned in the login status and whether the HTTP status code was 200.

7.3.2. Zero-Day Vulnerabilities Detection

After testing the vulnerable devices, the effectiveness of this work was initially explained. Next, we further demonstrate the effectiveness of the work. We chose different embedded devices than those that have been used to test brands. That is, various equipment was selected from the D-Link manufacturer for testing. Finally, we found four command injection vulnerabilities and three authentication bypass vulnerabilities in the D-Link series routers. It is worth mentioning that when testing the Netgear DGN2200 to cover existing vulnerabilities, a new authentication bypass vulnerability was discovered by EWVHunter. Our EWVHunter found seven zero-day vulnerabilities in five devices, five of which have applied for CVE numbers, and two were waiting for CVE official approval due to time. The vulnerability information found by EWVHunter is shown in Table 6.

Table 6. Summary of discovered vulnerabilities.

Device	Vul-ID	Vulnerability	Remotely Exploitable	CVE-ID
D-Link DIR-850L	1	Command injection	True	CVE-2019-17508
D-Link DIR-846	2	Command injection	True	CVE-2019-17509
	3	Command injection	True	unassigned
	4	Command injection	True	unassigned
D-Link DIR-412	5	Authentication bypass	True	CVE-2019-17511
D-Link DAP-1320	6	Authentication bypass	True	CVE-2019-17505
Netgear DGN2200	7	Authentication bypass	True	CVE-2019-17373

The data transmitted by D-Link DIR-850L is a XML type of POST. The input point on the web front-end is the NTP Server option. After user input, this program did not sanitize unsafe user-supplied data and passed it to a system shell, resulting in command injection vulnerability. D-Link DIR-850L's stateful network protocol belongs to the category of relevancy in context. We gave it a detailed vulnerability analysis in Section 7.2. The data transmitted by D-Link DIR-846 is a JSON type of POST. The first input point of the first vulnerability in the web front-end was where the wireless network name was set in "Wireless.html". As the user's input was not filtered, the page handle directly executed the transmitted string, which led to the appearance of a command injection vulnerability. As developers reused this part of the processing code, there were two similar vulnerabilities on the page "Guestwireless.html" and "MobileWireless.html". As we considered this situation, we designed a

fuzz scheduling, that is, the seeds that triggered the vulnerability would have the highest priority in the same fuzzing process. So EWVHunter could quickly find similar security flaws in the same device. The D-Link DIR-846 and the D-Link DIR-850L used the same vulnerability detection strategy.

The D-Link DIR-412 device's authentication bypass vulnerability was that the log information on the device could be seen when directly accessing the "log_get.php" page. The D-Link DAP-1320 device's authentication bypass vulnerability existed in the "uplink_info.xml" page, which could directly access and obtain the wireless authentication information Wi-Fi SSID and password without authentication. In most cases, the password of the wireless network was also the administrator's password, so the device could be controlled directly by using this vulnerability. We could gain unauthorized access to the Netgear DGN2200 because the device assumed this page did not require authentication if the end of the URL is "jpg". So, we could add "?x=1.jpg" at the vital URL end to view the information on the page without completing authentication. The detection strategy used for the above vulnerability was to verify response data and (200) status code.

7.4. Comparison with Existing Work

In this chapter, we will compare and analyze with WMIFuzzer, AFL, and boofuzz, and evaluate our work in terms of the ability and efficiency of fuzzing.

7.4.1. Ability to Fuzz Testing

We measured the superiority of the solution based on whether the solution supported the current protocol type and whether the encrypted transmission of the transmitted message was supported. To illustrate the advantage of our solution, we compared it with the three most relevant work: WMIFuzzer, AFL, and boofuzz. The specific comparison result is shown in Table 7. The True in the table indicates that the fuzzer could support it, and the False demonstrates that the fuzzer could not support it.

Table 7. Summary of the comparison of the ability to fuzz network protocols.

Type		EWVHunter	WMIFuzzer	AFL	Boofuzz
Stateful network protocol	Complexity of communication	True	False	False	False
	Relevancy in context	True	False	False	False
	Good transaction semantics	True	False	False	False
	State transition	True	False	False	False
Stateless network protocols		True	True	True	True
Transmission message encryption		True	False	False	False

7.4.2. Efficiency

Web front-end analysis module validity description. Our web front-end analysis module focused on finding the paths that could successfully trigger the network request event, and then build the activity path graph. It will be used to fire the network request events based on the graph in the fuzzing phase. The number of events related to network requests and the number of web front-end events are shown in Table 8. According to statistical analysis, it can be seen that not all events were related to network requests. This indicates that only part of the embedded web front-end logic was used to send network requests.

Table 8. Statistics for web front-end analysis of embedded web front-ends.

Device	#of Identified Network Events	Total#of Identified Events
Netgear DGN2200	622	916
Netgear DGN1000	1269	2500
Trendnet TEW-733GR	578	1893
Tenda AC9	162	179
D-Link DIR-850L	414	2005
D-Link DIR-846	2151	2367
D-Link DIR-412	256	1244
D-Link DAP-1320	201	333

Efficiency of vulnerability detection. We measured vulnerability detection efficiency based on the number of vulnerabilities found over time and the number of vulnerabilities found in test cases. To illustrate the effectiveness and efficiency of our scheme, we compared it with two of the most advanced fuzzers currently available: AFL [15] and boofuzz [16]. The original branch of AFL only targeted local binary program vulnerability detection. It could not directly perform fuzzing on the embedded device's communication protocol and could not remotely monitor the status of the device to determine whether the vulnerability was triggered. Therefore, we modified its code so that it could send mutated data and extend the AFL with the remote monitoring scheme used in EWVHunter. Boofuzz is another fuzzing for network protocols. It was a modified branch of Sulley, mainly to enhance the stability of Sulley. However, the launch of boofuzz required a data model. Each message was considered as a set of tokens separated by CR-LR or white space to generate data models in batches for testing [14]. AFL was used to test network protocols and required manual identification of where fuzz was needed. Both AFL and boofuzz required an initial seed to start, so we used the packets generated by EWVHunter as their initial seed. In addition, we used the knowledgebase built in this article to enhance their mutation patterns. It also allowed the patched AFL and boofuzz to run on the virtual machine with the same configuration for 24 h. To support testing, we modified AFL and boofuzz, and we called them AFL Patched and boofuzz Patched.

Table 9 shows the results of comparing EWVHunter with AFL Patched and boofuzz Patched. It could be seen that EWVHunter usually exceeded the popular open-source fuzzer in terms of test time and the number of security vulnerabilities discovered. EWVHunter could also detect vulnerabilities for devices with stateful network protocols, but AFL Patched and Boofuzz Patched were unable to do this. EWVHunter found a total of 12 vulnerabilities (seven command injection vulnerabilities and five authentication bypass vulnerabilities).

Table 9. Statistics on comparison of efficiency in finding vulnerabilities.

Vulnerability Type	Vul-CVE-ID	Device	EWVHunter	AFL Patched	Boofuzz Patched
Command injection	CVE-2017-6077	Netgear DGN2200	1 h 14 min	15 h 24 min	3 h 12 min
	CVE-2019-5071	Tenda AC9	56 min 45 s	3 h 42 min	42 min 12 s
	CVE-2019-5072		47 s	3 h 43 min	43 min 12 s
	CVE-2019-17508	D-Link DIR-850L	3 h 42 min	NA	NA
	CVE-2019-17509	D-Link DIR-846	2 h 11 min	NA	NA
	unassigned 1		8 min 27 s	NA	NA
	unassigned 2		7 min 44 s	NA	NA
Authentication bypass	N-day	Netgear DGN1000	28 min 33 s	19 h 6 min	13 h 29 min
	CVE-2018-7034	Trendnet TEW-733GR	45 min 01 s	21 h 17 min	18 h 6 min
	CVE-2019-17511	D-Link DIR-412	3 min 29 s	5 min 42 s	3 min 48 s
	CVE-2019-17505	D-Link DAP-1320	3 min 11 s	5 min 12 s	3 min 11 s
	CVE-2019-17373	Netgear DGN2200	25 min 17 s	17 h 30 min	52 min 18 s

We conducted an artificial analysis of the experimental results and found that there were several things worth paying attention to:

1. Neither the AFL nor the original versions of boofuzz were able to detect command injection and authentication bypass vulnerabilities. The main reason is that they did not have a detection strategy. After adding EWVHunter's remote detection strategies and using the request message generated by EWVHunter as a seed, AFL Patched and boofuzz Patched could detect these two kinds of vulnerabilities. However, without the knowledgebase built in this paper, AFL Patched and boofuzz Patched would difficult to find this vulnerability in 24 h;
2. The AFL patch and Boofuzz patch used the site map built by EWVHunter, so they could find authentication bypass vulnerabilities in D-Link DIR-412 and D-Link DAP-1320. Without the site map in the EWVHunter knowledgebase, they would not be able to find both interfaces. That shows that the knowledgebase constructed in this paper could effectively expand the breadth of fuzzing for embedded devices;
3. EWVHunter had the highest test efficiency. It took longer for AFL Patched to detect a vulnerability than boofuzz Patched, mainly because AFL Patched generated a lot of useless test data. Boofuzz Patched used the data model as a guide to generate structured messages. So the test cases generated by boofuzz Patched were more effective. Compared with AFL Patched and boofuzz Patched, EWVHunter filled data on the web front-end input sources. Therefore, most of the data generated were messages that the embedded device could accept and process. So our testing efficiency was higher;
4. For D-Link DIR-846 and DIR-850L, their protocol had status. Our EWVHunter could detect vulnerabilities, but AFL Patched and boofuzz Patched could not;
5. The two vulnerabilities in Tenda AC9 were on the same page and caused by similar reasons. So, the fuzz scheduling strategy in this paper could help us quickly find the vulnerability in the second location. Although the three vulnerabilities of D-Link DIR-846 were not on the same page, they could be quickly detected using the same type of payload. This shows that our fuzz strategy allowed EWVHunter to find the same type of vulnerability quickly after finding the first vulnerability.

8. Discussion

EWVHunter could effectively find vulnerabilities in embedded devices and support the fuzz testing of devices that implement stateful network protocol or encrypted transmission messages. However, there were still some ways for future improvements. In this section, we discussed the implications and the limitations of EWVHunter and muse about several ideas for future research directions.

Legitimacy Considerations and Coordinated Disclosure. We purchased these devices during our research without using online devices that existed on the Internet. After using EWVHunter to find the vulnerability and analyze the cause of the vulnerability, we immediately contacted the CVE organization. The CVE organization will contact the vendor to address these discovered security vulnerabilities.

Test Range of Devices. EWVHunter achieved the gray-box fuzz testing for the network protocol and added static analysis to extract information hidden in the firmware. So this paper focused on devices that had a web front-end and unpackable firmware. If EWVHunter wanted to be applied to IoT devices, IoT devices must have a web front-end and unpackable firmware. Although a large percentage of devices have the above characteristics, not every (IoT) device has it. For example, a smart bracelet with limited hardware and power resources does not have a web front-end. However, accessing the device is complicated in this case, so it is difficult to attack in practice successfully.

Discovery of Device Backdoor. EWVHunter was unable to detect hidden backdoors in the device and treats mathematically based backdoors (for example, "the password must be an integer divisible by 10") as "correct" authentication in particular. EWVHunter needed to perform a full static analysis

during the firmware analysis phase to detect such a backdoor. Nevertheless, the security static analysis tool for embedded devices with diverse page handle structures (such as PHP, CGI-bin, and LUA) does not exist. Moreover, with the discovery of mathematical type backdoors, this ability will involve additional complexity related to constraint solving, and we believe that this analysis is beyond the scope of this study.

Limitations of Vulnerability Detection. As general embedded devices do not have local monitoring functions, EWVHunter needs to monitor remotely whether some vulnerabilities are triggered. The primary focus is automatically detecting the authentication bypass vulnerability and command injection vulnerability. The command injection vulnerability can allow the attacker to take complete control of an embedded device, and the authentication bypassing vulnerability could leak information. If the interface with the authentication bypass vulnerability also has an input point for the command injection vulnerability or the interface with the authentication bypass vulnerability leaks the login information, it will seriously threaten the embedded device security. The other types of vulnerabilities detection in embedded devices, such as XSS vulnerabilities, is still a future work. The reason why this article does not set a strategy to detect the target's binary vulnerability is that the remote monitoring method currently used by others to detect the crash of the target device is still inaccurate. There is no way to detect a binary vulnerability that does not cause the device to restart or crash. Therefore, we will introduce the monitoring of the device state in future work to ensure that binary vulnerability can be found well.

The Accuracy of the Results. The most prone to false positives in this article is the detection of vulnerabilities in identity authentication. This article uses the most straightforward status code analysis for authentication bypass vulnerability detection. If the interface can be accessed without authentication, then we consider it to be vulnerable. This detection strategy treats certain interfaces that do not require authentication as vulnerabilities, and EWVHunter will incorrectly report them. EWVHunter has no way to judge how much harm the authentication bypass vulnerability can cause, and its risk assessment needs further manual analysis. Furthermore, we want to combine authentication bypass vulnerabilities with command injection vulnerabilities to increase the risk level of vulnerability.

Discovery of Homology Vulnerabilities in Embedded Devices. As the firmware in similar types of embedded devices is often similar to each other, the homology analysis can be used to analyze this series of the firmware after using the fuzz testing method proposed in this paper to find a security vulnerability in a device. Then, we can find the same vulnerability in different models of embedded devices. Luo et al. [49] implemented GeneDiff, which can perform homology analysis across architectures to analyze embedded device firmware so that vulnerabilities in similar models can be discovered.

9. Conclusions

We proposed the first gray-box fuzzing framework targeting embedded devices firmware that utilizes the web front-end to detect authentication bypass and command injection vulnerabilities. To achieve efficient fuzzing, we designed a set of technologies including constructive knowledgebase to guide fuzz testing, automatic analysis for web front-end to build an activity path graph, filling the data at the web front-end input sources, and remote vulnerability detection strategies to monitor devices. Through conducting experiments on eight real-world embedded devices, we successfully identified 12 vulnerabilities including seven zero-day vulnerabilities. These zero-day vulnerabilities discovered by EWVHunter could affect 31,996 online embedded devices.

Author Contributions: E.W. contributed to the design of the study, conducted the experiments, analyzed the results, and wrote the manuscript. W.X. contributed the IDEA, Z.W. implemented some experiments, B.W., Z.L., T.Y. proofread the paper. All authors have read and agreed to the published version of the manuscript.

Acknowledgments: We would like to sincerely thank all the reviewers for your time and expertise on this paper. Your insightful comments help us improve this work.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

IoT	Internet of Things
CVE	Common Vulnerabilities and Exposures
CSS	Cascading Style Sheets
OOB	Out-of-band
BAT	Binary Analysis Toolkit
XSS	Cross-site Scripting
SSL	Secure Sockets Layer
HTTP	HyperText Transfer Protocol
URL	Uniform Resource Locator
SNMP	Simple Network Management Protocol
FTP	File Transfer Protocol
BGP	Border Gateway Protocol
SMB	Server Message Block
DDoS	Distributed Denial-of-service
COTS	Commercial Off-the-shelf
UI	User Interface
CSRF	Cross-site Request Forgery
AFL	American Fuzzy Lop
PHP	Hypertext Preprocessor
CGI	Common Gateway Interface
CPU	Central Processing Unit

References

1. Gartner Identifies Top 10 Strategic IoT Technologies and Trends. Available online: https://www.gartner.com/en/newsroom/press-releases/2018-11-07-gartner-identifies-top-10-strategic-iot-technologies-and-trends (accessed on 7 November 2018).
2. O'Donnell, L. 2 Million IoT Devices Vulnerable to Complete Takeover. Available online: https://threatpost.com/iot-devices-vulnerable-takeover/144167/ (accessed on 29 April 2019).
3. Costin, A. Security of cctv and video surveillance systems: Threats, vulnerabilities, attacks, and mitigations. In Proceedings of the 6th International Workshop on Trustworthy Embedded Devices, Vienna, Austria, 28 October 2016; pp. 45–54.
4. Securing IoT Devices: How Safe Is Your Wi-Fi Router? Available online: https://www.theamericanconsumer.org/wp-content/uploads/2018/09/FINAL-Wi-Fi-Router-Vulnerabilities.pdf (accessed on 28 September 2018).
5. Angrishi, K. Turning internet of things (iot) into internet of vulnerabilities (iov): Iot botnets. *arXiv* **2017**, arXiv:1702.03681.
6. State of the Internet Security Spotlight Iot Rise of 300 gbp Ddos Attacks. 2017. Available online: https://www.akamai.com/cn/zh/multimedia/documents/social/q4-state-of-the-internet-security-spotlight-iot-rise-of-300-gbp-ddos-attacks.pdf (accessed on 3 February 2017).
7. Costin, A.; Zaddach, J.; Francillon, A.; Balzarotti, D. A large-scale analysis of the security of embedded firmwares. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 95–110.
8. Chen, D.D.; Woo, M.; Brumley, D.; Egele, M. *Towards Automated Dynamic Analysis for Linux-Based Embedded Firmware*; NDSS: New York, NY, USA, 2016; Volume 16, pp. 1–16.
9. Craig, H. Binwalk: Firmware Analysis Tool. 2019. Available online: https://github.com/ReFirmLabs/binwalk (accessed on 14 May 2020).
10. Hemel, A.; Kalleberg, K.T.; Vermaas, R.; Dolstra, E. Finding software license violations through binary code clone detection. In Proceedings of the 8th Working Conference on Mining Software Repositories, Honolulu, HI, USA, 21–28 May 2011; pp. 63–72.

11. Cui, A.; Costello, M.; Stolfo, S. *When Firmware Modifications Attack: A Case Study of Embedded Exploitation*; NDSS: San Diego, CA, USA, 2013; pp. 1–13.
12. Gascon, H.; Wressnegger, C.; Yamaguchi, F.; Arp, D.; Rieck, K. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*; Springer: Berlin, Germany, 2015; pp. 330–347.
13. Zaddach, J.; Bruno, L.; Francillon, A.; Balzarotti, D. *AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares*; NDSS: New York, NY, USA, 2014; Volume 14, pp. 1–16.
14. Wang, D.; Zhang, X.; Chen, T.; Li, J. Discovering Vulnerabilities in COTS IoT Devices through Blackbox Fuzzing Web Management Interface. *Secur. Commun. Netw.* **2019**, *2019*, 5076324. [CrossRef]
15. Zalewski, M. American Fuzzy Lop. 2014. Available online: https://github.com/google/AFL (accessed on 14 May 2020).
16. Pereyda, J. A Fork and Successor of the Sulley Fuzzing Framework. 2020. Available online: https://github.com/jtpereyda/boofuzz (accessed on 4 June 2020).
17. Shoshitaishvili, Y.; Wang, R.; Hauser, C.; Kruegel, C.; Vigna, G. *Firmalice-Automatic Detection of Authentication Bypass VUlnerabilities in Binary Firmware*; NDSS: New York, NY, USA, 2015.
18. A1-Injection. Available online: https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A1-Injection (accessed on 6 June 2020).
19. Bau, J.; Bursztein, E.; Gupta, D.; Mitchell, J. State of the art: Automated black-box web application vulnerability testing. In Proceedings of the 2010 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, USA, 16–19 May 2010; pp. 332–345.
20. Costin, A.; Zarras, A.; Francillon, A. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, Xi'an, China, 30 May–3 June 2016; pp. 437–448.
21. Dahse, J.; Holz, T. *Simulation of Built-in PHP Features for Precise Static Code Analysis*; NDSS: New York, NY, USA, 2014; Volume 14, pp. 23–26.
22. Chipounov, V.; Kuznetsov, V.; Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM Sigplan Not.* **2011**, *46*, 265–278. [CrossRef]
23. Aitel, D. The Advantages of Block-Based Protocol Analysis for Security Testing. 2002. Available online: https://pdfs.semanticscholar.org/3cfc/a42b42a22a23d5f13d27b9bf3e8a1dec98bb.pdf?_ga=2.122221003.1719363065.1591757503-1908025882.1571047123 (accessed on 10 April 2020).
24. Kaksonen, R.; Laakso, M.; Takanen, A. Software security assessment through specification mutations and fault injection. In *Communications and Multimedia Security Issues of the New Century*; Springer: Berlin, Germany, 2001; pp. 173–183.
25. Chunlei, W.; Li, L.; Qiang, L. Automatic fuzz testing of web service vulnerability. In Proceedings of the 2014 International Conference on Information and Communications Technologies (ICT 2014), Nanjing, China, 15–17 May 2014.
26. Mendez, X. Wfuzz—The Web Fuzzer. Available online: https://github.com/xmendez/wfuzz (accessed on 9 May 2020).
27. Banks, G.; Cova, M.; Felmetsger, V.; Almeroth, K.; Kemmerer, R.; Vigna, G. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *International Conference on Information Security*; Springer: Berlin, Germany, 2006; pp. 343–358.
28. 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, London, UK, 11–15 November 2019; pp. 2525–2527.
29. Ma, R.; Wang, D.; Hu, C.; Ji, W.; Xue, J. Test data generation for stateful network protocol fuzzing using a rule-based state machine. *Tsinghua Sci. Technol.* **2016**, *21*, 352–360. [CrossRef]
30. Tsankov, P.; Dashti, M.T.; Basin, D. SECFUZZ: Fuzz-testing security protocols. In Proceedings of the 2012 7th International Workshop on Automation of Software Test (AST), Zurich, Switzerland, 2–3 June 2012; pp. 1–7.
31. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Lin, Z.; Wang, X.; Lau, W.C.; Sun, M.; Yang, R.; Zhang, K. *IoTFuzzer: Discovering Memory Corruptions in IoT through App-Based Fuzzing*; NDSS: New York, NY, USA, 2018.
32. Barth, A.; Jackson, C.; Mitchell, J.C. Robust defenses for cross-site request forgery. In Proceedings of the 15th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 27–31 October 2008; pp. 75–88.

33. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. *Driller: Augmenting Fuzzing through Selective Symbolic Execution*; NDSS: New York, NY, USA, 2016; Volume 16, pp. 1–16.
34. Yue, T.; Tang, Y.; Yu, B.; Wang, P.; Wang, E. LearnAFL: Greybox Fuzzing With Knowledge Enhancement. *IEEE Access* **2019**, *7*, 117029–117043. [CrossRef]
35. Lee, J.Y.; Lin, W.C.; Huang, Y.H. A lightweight authentication protocol for internet of things. In Proceedings of the 2014 International Symposium on Next-Generation Electronics (ISNE), Kwei-Shan, Taiwan, 7–10 May 2014; pp. 1–2.
36. Poh, G.S.; Gope, P.; Ning, J. PrivHome: Privacy-preserving authenticated communication in smart home environment. *IEEE Trans. Dependable Secure Comput.* **2019**. [CrossRef]
37. Sain, M.; Kang, Y.J.; Lee, H.J. Survey on security in Internet of Things: State of the art and challenges. In Proceedings of the 2017 19th International conference on advanced communication technology (ICACT), Bongpyeong, Korea, 19–22 February 2017; pp. 699–704.
38. Prescatore, J; Gartner. Quoted in ComputerWorld. Available online: http://www.computerworld.com/printthis/2005/0,4814,99981,00.html (accessed on 20 June 2005).
39. Mitchell, L.J. *PHP Web Services: APIs for the Modern Web*; O'Reilly Media Inc.: Sebastopol, CA, USA, 2016.
40. Guelich, S.; Gundavaram, S.; Birznieks, G. *CGI Programming with Perl: Creating Dynamic Web Pages*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2000.
41. Hester, A.; Borges, R.; Ierusalimschy, R. Building flexible and extensible web applications with Lua. *J. Univers. Comput. Sci.* **1998**, *4*, 748–762.
42. Cyberspace Asset Search Engine. Available online: https://fofa.so/ (accessed on 10 June 2020).
43. Doupé, A.; Cova, M.; Vigna, G. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*; Springer: Berlin, Germany, 2010; pp. 111–131.
44. Dictionary of Attack Patterns and Primitives for Black-Box Application Fault Injection and Resource Discovery. Available online: https://github.com/fuzzdb-project/fuzzdb (accessed on 27 February 2020).
45. A List of Useful Payloads and Bypass for Web Application Security and Pentest/CTF. Available online: https://github.com/swisskyrepo/PayloadsAllTheThings (accessed on 6 June 2020).
46. Chromium is an Open-Source Browser Project that Aims to Build a Safer, Faster, and More Stable Way for all Internet Users to Experience the Web. 2020. Available online: https://www.chromium.org/Home (accessed on 9 June 2020).
47. Headless Chrome/Chromium Automation Library. Available online: https://github.com/miyakogi/pyppeteer (accessed on 8 May 2020).
48. Vtest is Used to Assist Security Engineers in Mining, Testing Vulnerabilities. Available online: https://github.com/opensec-cn/vtest (accessed on 21 July 2019).
49. Luo, Z.; Wang, B.; Tang, Y.; Xie, W. Semantic-Based Representation Binary Clone Detection for Cross-Architectures in the Internet of Things. *Appl. Sci.* **2019**, *9*, 3283. [CrossRef]