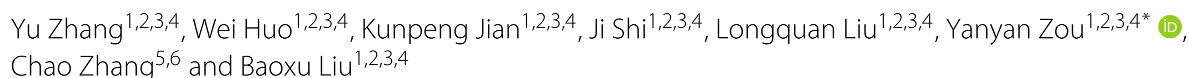
Zhang et al. Cybersecurity (2021) 4:24
https://doi.org/10.1186/s42400-021-00091-9

Cybersecurity

RESEARCH Open Access

ESRFuzzer: an enhanced fuzzing framework for physical SOHO router devices to discover multi-Type vulnerabilities



Yu Zhang[1,2,3,4], Wei Huo[1,2,3,4], Kunpeng Jian[1,2,3,4], Ji Shi[1,2,3,4], Longquan Liu[1,2,3,4], Yanyan Zou[1,2,3,4*], Chao Zhang[5,6] and Baoxu Liu[1,2,3,4]

Abstract

SOHO (small office/home office) routers provide services for end devices to connect to the Internet, playing an important role in cyberspace. Unfortunately, security vulnerabilities pervasively exist in these routers, especially in the web server modules, greatly endangering end users. To discover these vulnerabilities, fuzzing web server modules of SOHO routers is the most popular solution. However, its effectiveness is limited due to the lack of input specification, lack of routers' internal running states, and lack of testing environment recovery mechanisms. Moreover, existing works for device fuzzing are more likely to detect memory corruption vulnerabilities.

In this paper, we propose a solution ESRFuzzer to address these issues. It is a fully automated fuzzing framework for testing physical SOHO devices. It continuously and effectively generates test cases by leveraging two input semantic models, i.e., KEY-VALUE data model and CONF-READ communication model, and automatically recovers the testing environment with power management. It also coordinates diversified mutation rules with multiple monitoring mechanisms to trigger multi-type vulnerabilities. With the guidance of the two semantic models, ESRFuzzer can work in two ways: general mode fuzzing and D-CONF mode fuzzing. General mode fuzzing can discover both issues which occur in the CONF and READ operation, while D-CONF mode fuzzing focus on the READ-op issues especially missed by general mode fuzzing.

We ran ESRFuzzer on 10 popular routers across five vendors. In total, it discovered 136 unique issues, 120 of which have been confirmed as 0-day vulnerabilities we found. As an improvement of SRFuzzer, ESRFuzzer have discovered 35 previous undiscovered READ-op issues that belong to three vulnerability types, and 23 of them have been confirmed as 0-day vulnerabilities by vendors. The experimental results show that ESRFuzzer outperforms state-of-the-art solutions in terms of types and number of vulnerabilities found.

Keywords: Fuzzing, IoT, Automatic vulnerability detection

*Correspondence: zouyanyan@iie.ac.cn
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
Full list of author information is available at the end of the article

© The Author(s). 2021 **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

https://t.me/learningnets

Introduction

More and more end devices, such as laptops, pads, smartphones, smart-home devices, and wearable devices, are used in social life. They are usually connected to the Internet through small office and home office (SOHO) routers. As a result, SOHO routers are in a prominent position that isolates end users from external Internet (Chen et al. 2016) and processes end users' all traffic. The security of SOHO routers is much more critical than ever.

Unfortunately, the security vulnerabilities pervasively exist in SOHO routers (CERT 2016), (Khandelwal 2018). According to a recent report (ACI 2018), 83% of popular routers contain vulnerable code. These vulnerabilities are one of the essential exploiting targets by adversaries. In 2018, Cisco Talos found malware *VPNFilter* targeted to Linksys, MikroTik, NETGEAR, and TP-Link networking equipments, which are all SOHO routers, and infected at least 500,000 devices in at least 54 countries (Largent and New VPNFilter malware targets at least 500K networking devices worldwide 2018). Besides, the leading exploit acquisition platform Zerodium (2015) has added the requirement for routers in 2018. Therefore, discovering vulnerabilities in SOHO routers becomes significantly important.

A typical architecture of the SOHO router is shown on the right side of Fig. 1. As network equipment, the SOHO router provides networking service, e.g., routing, for the end devices connected to it. More importantly, it also leverages web services for administration and configuration, due to its lack of user interface, e.g., keyboard, video, mouse. These web services are provided by some widely used protocols, e.g., HyperText Transfer Protocol (HTTP). We call these protocols as management protocols in this paper.

A typical management protocol is implemented by embedding a web server (backend) into the original device. Usually, web servers in different routers are customized by device vendors and are more vulnerable. Recent works (Costin et al. 2016), (Chen et al. 2016), (Chen et al. 2018) shown that most of the SOHO router vulnerabilities identified are be associated with web services, such as command injection vulnerabilities in PHP server-side scripts and memory corruption vulnerabilities in processing mobile application web requests. Therefore, this paper also focuses on discovering vulnerabilities of the web server of the SOHO router.

Fuzz testing (i.e., fuzzing) is considered to be a powerful technique to discover vulnerabilities. However, there is little research on fuzzing web server of SOHO router (FWSR), except IoTFuzzer (Chen et al. 2018), an app-based fuzzing framework. With the help of the program logic of mobile APP that could control the device, the approach produces meaningful test cases and triggers device bugs. Although it has partially solved the problems of FWSR, in general, FWSR remains challenging.

Challenge 1: Fuzzing Dedicated System

As an embedded device, the function of the SOHO router is dedicated and cannot be extended easily. Therefore, the internal running state cannot be acquired directly during fuzzing, and its normal running is hard to restore when the device is made to be stuck during testing. These make it difficult for FWSR continuously and effectively. Although fuzzing based on emulation is a promising way to obtain internal executing information that can guide fuzzing, it is limited in emulating various routers in full-system mode. (See "Related work" section)

Challenge 2: Analyzing Input Semantics

The input of web server is accordant with standard protocols, e.g., HyperText Transfer Protocol (HTTP), however, their internal data that encode the exchanging information are in various formats across routers. Without investigating the format of the internal data and information exchanging process, FWSR is not effective in terms of code coverage.

Challenge 3: Discovering Multi-type Vulnerabilities

As aforementioned, the web server is almost customized by vendors from the frontend to the backend. Therefore not only web vulnerabilities, such as command injection and cross-site scripting, but also memory corruption, should be discovered by FWSR. Unfortunately, different types of vulnerability need to be triggered by different payloads, as well as monitored by different methods. Such requirements make the design of FWSR more difficult.

Our Approach.

In the preliminary version of this paper, we propose an automatic fuzzing framework SRFuzzer (Zhang et al. 2019) for FWSR based on physical devices to address the above challenges. It drives the fuzzing process continuously by automatic seed generation and automatic power control. To model the input semantics, it leverages two models to constrain test cases, i.e., the KEY-VALUE data model (K-V model for short) to describe the format of internal data of requests, and the CONF-READ communication model (C-R model for short) to describe the temporal sequence of requests. Moreover, our framework coordinates different mutation rules with multiple monitoring mechanisms to effectively trigger four types of vulnerabilities, i.e., vulnerabilities of *memory corruption*, *command injection*, *cross-site scripting (XSS)* and *information disclosure*. To the best of our knowledge, it is the first whole-process fully-automatic framework for FWSR. Furthermore, by a little human effort on collecting web requests and monitoring running state, it could outperform fully-automatic fuzzing. We have implemented a prototype of our solution SRFuzzer and deployed it in a real-world environment. To evaluate its effectiveness and generality, we ran SRFuzzer

https://t.me/learningnets

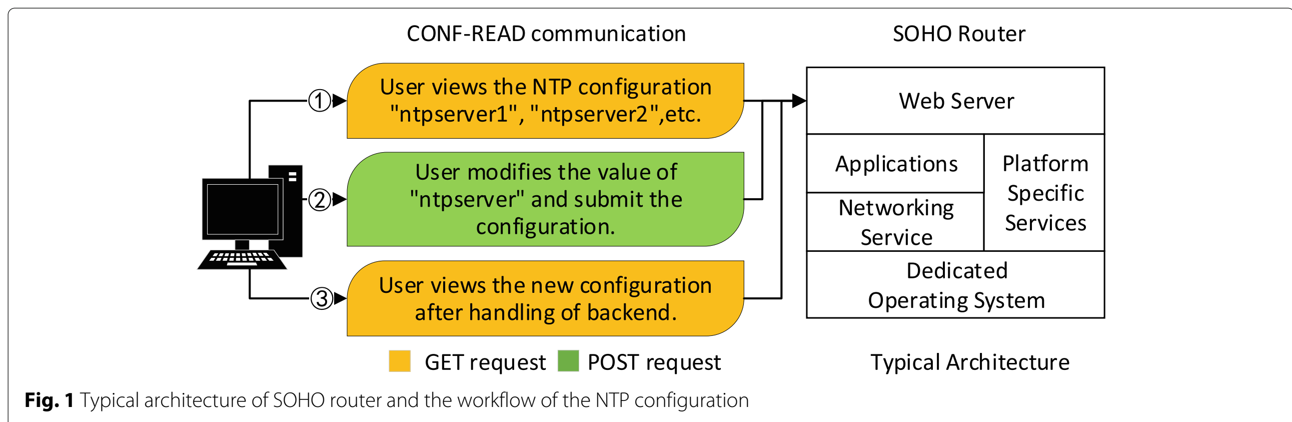


Fig. 1 Typical architecture of SOHO router and the workflow of the NTP configuration

on 10 popular routers across 5 vendors. Benefit from the comprehensive new monitoring method, we have got 208 unique exceptional behaviors. Almost half of the exceptional behaviors are confirmed as 101 unique issues that belong to the aforementioned four vulnerability types. After responsible disclosing vulnerabilities to the corresponding vendors, we obtained total 97 assigned IDs, i.e., 43 CVE[1] IDs, 52 PSV[2] IDs and 2 CNVD[3] IDs[4].

According to the CONF-READ model and previously discovered issues, the vulnerability triggered in a CONF operation (CONF-op issue) can always be triggered with only one request, while the vulnerability triggered in a READ operation (the READ-op issue for short) needs two requests. This is because READ-op issues are related to a proper CONF operation which is related to the same k-v pairs. For a vulnerable k-v pair, only its mutated value is configured by a CONF operation, then the vulnerability could be triggered in a READ operation. SRFuzzer triggers the READ-op issue by crafting a multi-phase communication. However, this method could miss some issues if the vulnerable k-v pair is not configured in the previous CONF operation. Obviously, the CONF operation limits the detection of READ-op issues for the lack of a proper configuration.

Besides the explicit way to execute the CONF operation with crafted requests through the official way by management protocol, we found an implicit way to achieve it by using the “backup and restore configuration” feature of the device. This feature usually provides convenience to users to restore a misconfigured or new device. The configuration of SOHO routers always contains a set of key-value pairs and can exist in many ways, such as NVRAM or a database. It also can export as a file from the device for backup or configuration restore. We found out the configuration restoration can be treated as a set of CONF operations and always with weak security checks during our research by reverse-engineering firmware implementation. In other words, a specially crafted configuration file can execute the CONF operation for arbitrary k-v pairs with no or few validity checks comparing with the previous CONF operation of SRFuzzer. Besides, we can also use the command (“Motivation” section shows an example) to achieve the same result in some devices to obtain the shell. We call them Direct CONF operation (D-CONF operation) in order to distinguish it from the normal CONF operation. With D-CONF operation, we can detect READ-op issues more efficiently and accurately.

In this paper, we present ESRFuzzer (short for an Enhanced SOHO Router Fuzzing Framework), an enhancement of SRFuzzer, to improve the effectiveness of SRFuzzer in three major aspects:

(i) Besides generating the seed for general mode fuzzing through the web page by crawlers, ESRFuzzer can generate seeds for D-CONF operation by parsing READ handlers of the backend and obtaining related k-v pairs of a READ handler through static program analysis.
(ii) It proposes a novel D-CONF mode fuzzing, guided by the enhanced CONF-READ model, to specifically trigger the READ-op issue by configuring the k-v pair directly with D-CONF operation. In this mode of fuzzing, the ESRFuzzer can focus on triggering READ-op issues without care of the construction of corresponding CONF operations.
(iii) Based on the cooperation of D-CONF operation and READ operation, several previously missed issues can be detected efficiently and accurately. These issues contain memory corruption, command injection, and stored cross-site scripting.

We have implemented ESRFuzzer as a prototype tool by integrating the above improvements into SRFuzzer and deployed it in a real-world environment. To evaluated its effectiveness, we ran ESRFuzzer on 7 routers

[1] Common Vulnerabilities and Exposures.
[2] Vulnerability identifications of assigned by the vendor NETGEAR.
[3] China Nation Vulnerability Database.
[4] All CVE IDs, PSV IDs, and CNVD IDs are listed in the appendix.

https://t.me/learningnets

which support the D-CONF mode fuzzing from the 10 routers SRFuzzer had tested. With the help of the novel fuzzing method, we have discovered 35 previous undiscovered issues that belong to 3 vulnerability types.

In summary, this paper makes the following main improvements:

- We analyze the implementation of the "backup and restore configuration" feature of several popular SOHO routers of different vendors by reverse engineering. Then we find out the D-CONF operation to execute the CONF operation for arbitrary k-v pairs with no or few validity checks.
- We extend the C-R model with the Direct CONF operation and propose a novel method to discover more issues triggered in READ operation, while these issues could be missed in the traditional method.
- We extend the SRFuzzer by supporting D-CONF mode fuzzing, which contains the Direct CONF operation and corresponding READ operation to find more READ-op issues automatically.
- We evaluated ESRFuzzer over 7 real-world SOHO routers in D-CONF mode fuzzing. In total, it successfully discovered 35 confirmed issues previously missed. After we manually completed all of the PoCs and reported them, 23 issues have been confirmed as 0-day vulnerabilities by vendors.

This paper is organized as follows: "Motivation" section presents our motivations and insights to overcome the challenges. "Detailed design" section overviews ESRFuzzer and describes the detailed design. "Experiment and evaluation" section introduces the experiments and evaluation. "Case study" section describe an interesting and concrete real-world case. The shortcomings of our framework and related work are discussed in "namerefsection5" and "Related work" sections. At last, we concludes in "Conclusion" section.

Motivation

The design goal of ESRFuzzer is to build an automatic fuzzing framework for FWSR and to find as many vulnerabilities as possible. It is straightforward to build the framework based on firmware emulation. However, such a solution is extremely hard in general because of the diversity in various dedicated component built-in routers. We argue that it would be a general framework to automatically fuzz the physical SOHO router directly, while it is still challenging.

In this section, we first introduce the vulnerabilities we focus on as well as the assumptions of our environment. Then we summarize design challenges in fuzzing real-world routers automatically as well as in depth. We analyze the root cause of the router vulnerabilities based on C-R and K-V models with a motivating example.

Scope and assumptions

Besides networking services, web services are also embodied into SOHO routers for the sake of administration and configuration. They are usually provided by standard protocols, including HyperText Transfer Protocol (HTTP), Simple Object Access Protocol (SOAP), Universal Plug and Play (UPnP) and more. These protocols are implemented by vendors and are used to set Wi-Fi password, to find end devices, etc. We call these protocols as management protocols and focus on HTTP protocol in this paper. Moreover, we suppose SOHO routers use built-in WAN/LAN web services, other than mobile-to-web services (Chen et al. 2018), as their default settings.

A typical management protocol implementation of a SOHO router consists of three major parts, i.e., a frontend, a backend, and a database. The frontend shows the current settings of the router and guides the user for configuration. The backend parses the requests received from the frontend and configures related services. The database stores the current configuration in several places, such as NVRAM, databases, and configuration files. Figure 2 describes the workflow of these parts in terms of CONF and READ operations.

There are three operations in the workflow: **Normal CONF operation** (CONF operation), **READ operation** and **Direct CONF operation** (D-CONF operation). During Normal CONF operation, the request generated in the frontend will be parsed by CONF handler of the backend. During READ operation, the READ handler of the backend generates the corresponding information that is displayed in the frontend. These two operations are common in almost all SOHO routers. There is also a special CONF operation called Direct CONF operation. In this way, the configuration can be set through *shell command* or *restore configuration* feature without being processed by the CONF handler.

Both the Normal CONF operation and D-CONF operation can config the device database which can be read by READ operation. However, the Normal CONF operation is more general than D-CONF operation among almost all devices while the D-CONF operation can operate more accurately.

Figure 2 also shows four typical types of vulnerabilities in the workflow. Vulnerabilities of memory corruption and command injection often occur in the backend, while XSS often occurs in the frontend. Information disclosure vulnerability may occur in both the frontend and backend. Different types of vulnerability can be triggered by various causes, e.g., memory corruption is usually triggered

https://t.me/learningnets

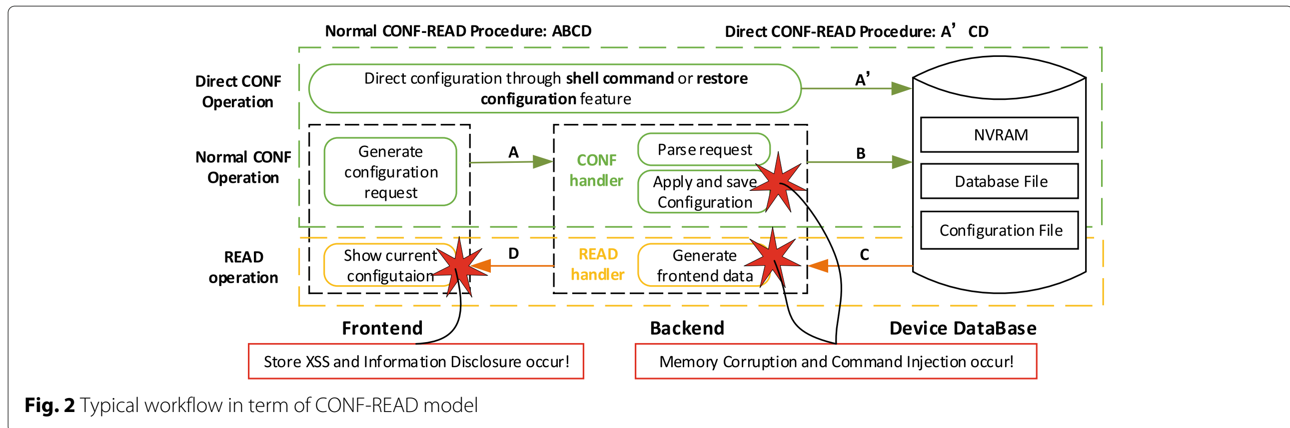


Fig. 2 Typical workflow in term of CONF-READ model

by improper user's input handling. Information disclosure in this paper refers to the access of privileged data without appropriate permission. The other two vulnerabilities are common web vulnerabilities, and most XSS issues are stored XSS in routers.

Automating the whole fuzzing process

The most important steps for automating the FWSR are input generation and running restoration.

As aforementioned, the web server is implemented differently across device vendors, which means the format of the request to each server is also different. So we choose mutation-based fuzzing since it is hard to adopt general generation-based fuzzing. In order to automatically collect seeds, which is essential for mutation-based fuzzing, we design a crawler to get as many requests as possible. Meanwhile, interactions with web services should be carefully dealt with to keep the server running. These requests are mutated by specific rules, which will be discussed in "Fuzzing in depth" section, and are fed to a physical SOHO router.

Once a specific testcase makes the server failure, ESRFuzzer should observe the occurrence and drive the fuzzing into the next test. Normally, it is relatively trivial to restart or resume a crashed process when fuzzing software on a standard computer system. On the contrary, it is not easy to reset or restart a stuck SOHO router without human interference. The router falls into "zombie" state during the fuzzing process mainly for two reasons. Firstly, there might be no response sent back when the process of the web server is crashed by a malformed request. Secondly, self-protection mechanism of some devices will forbid the access to the web server if specific exceptional conditions are met.

In order to restart the web server automatically, we leverage power control equipment to manage the SOHO router. Specifically, we use the smart plug which is widely used nowadays. Each smart plug powers a router and is controlled by ESRFuzzer. It is connected to ESRFuzzer through Wi-Fi. Once ESRFuzzer has monitored a stuck router, it sends the plug restarting command by plug's internal APIs, and consequently restarts the device. Then the web server could be recovered from the crashed state.

Fuzzing in depth

Coverage-guided fuzzing techniques, such as AFL (Zalewski 2014), have largely enhanced the traditional mutation-based fuzzing. Due to the requirement of instrumentation on code, they are infeasible for real-world dedicated devices. Moreover, the root cause of the vulnerabilities of routers is data inconsistency during the request processing, which makes the effectiveness of FWSR hardly be improved by those techniques. We design a KEY-VALUE data model and a CONF-READ communication model to describe the semantics of requests, i.e., the format of internal data in requests and the relations between different requests, and to guide the design of mutation rules. A motivating example which contains 3 vulnerabilities is presented to illustrate these models in a more concrete manner.

Motivating Example. To illustrate the design challenges of mutation rules, we present a configuration process for network time protocol (NTP) by HTTP requests. The left half of Fig. 1 shows the general process, and Fig. 3 shows the backend handling procedures related to the process.

There are usually three steps to configure the NTP option of a router:

(i) The administrator of router sends an HTTP GET request to access the configuration web page through a URL, e.g., "http://192.168.0.1/apply.cgi/NTP_debug.htm". He/she also gets the current domain name of `ntpserver1`, which is used by `read_ntpserver1()` shown in Line 9–13 of Fig. 3. We call this process READ operation.
(ii) The administrator modifies the domain name by sending an HTTP POST request. The new domain name of `ntpserver1` is submitted and stored into

https://t.me/learningnets

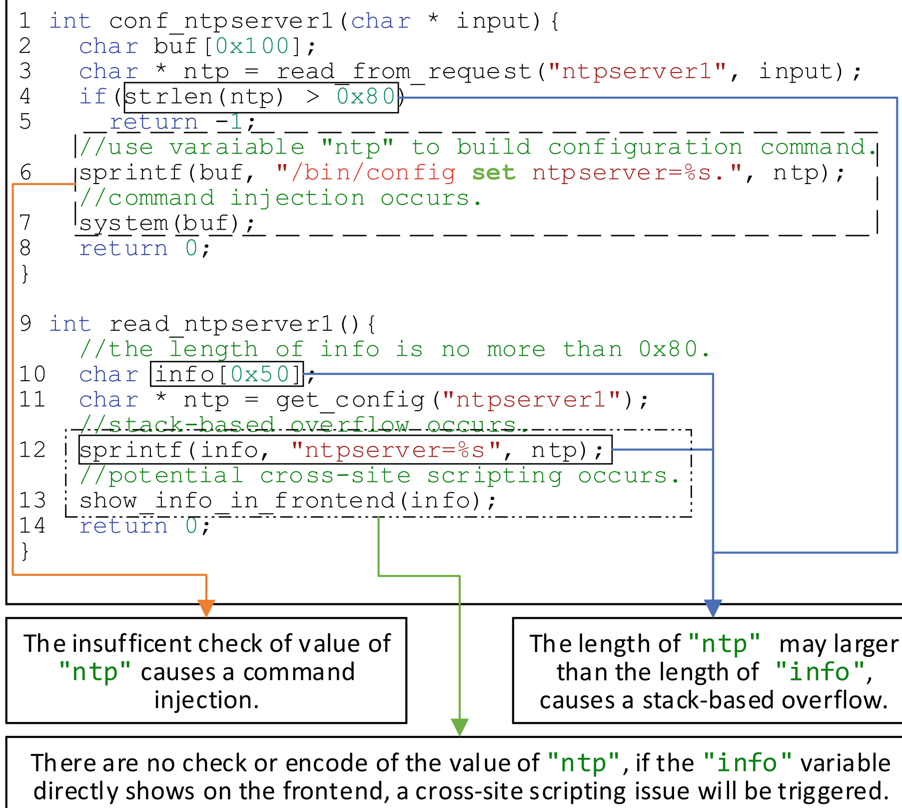


Fig. 3 Backend Handlers Code Snippet of READ and CONF Operation for NTP configuration

the database through the backend procedure `conf_ntpserver1()`. We call this process CONF operation.

(iii) He/She checks whether the newly submitted domain name is configured correctly by another READ operation.

However, there are 3 different types of vulnerability in this code snippet. We will explain these issues with KEY-VALUE data model and CONF-READ communication model.

KEY-VALUE Data Model. Supposing configuration process is now in CONF operation and the raw request is shown on top half of Fig. 4. The backend deals with the request as shown in Line 3, 6, 7 of Fig. 3 and a *command injection* issue occurs. Function `conf_ntpserver1()`, a CONF handler in the backend, would match the string “ntpserver1” from the request then get a domain name. The domain name is used as an argument of `/bin/config` command which would be executed by `system()` in turn. However, there would be a command injection if the request contains a string like “ntpserver1=;reboot;”, and the shell command “reboot;” would be invoked after executing `/bin/config` command.

The input triggering this vulnerability requires two conditions, i.e., the string “ntpserver1=” keeps unchanged and the format of `config` command keeps valid. Therefore, performing the randomized mutation on raw requests to generate test cases is probably meaningless in this case.

In order to generate meaningful test cases in both READ and CONF operation, we design the KEY-VALUE data model to describe the constraints on a single request. Each request should be composed of key-value pairs (k-v pairs for short), of which the key stands for the variable name like “ntpserver1” and keeps unchanged. The value could be assigned to the variable, so it should be consistent with the type requirement of the key. For example, if a key requires “domain name”, the value should be precisely a domain name. Although raw requests might be in other data formats, e.g. JSON and XML, the model still works well.

We describe the consistency between key and value by labeling attributes on the k-v pair according to the value. We summarize three types of attributes: *number*, *fixed string* and *variable string*. These attributes guide the

https://t.me/learningnets

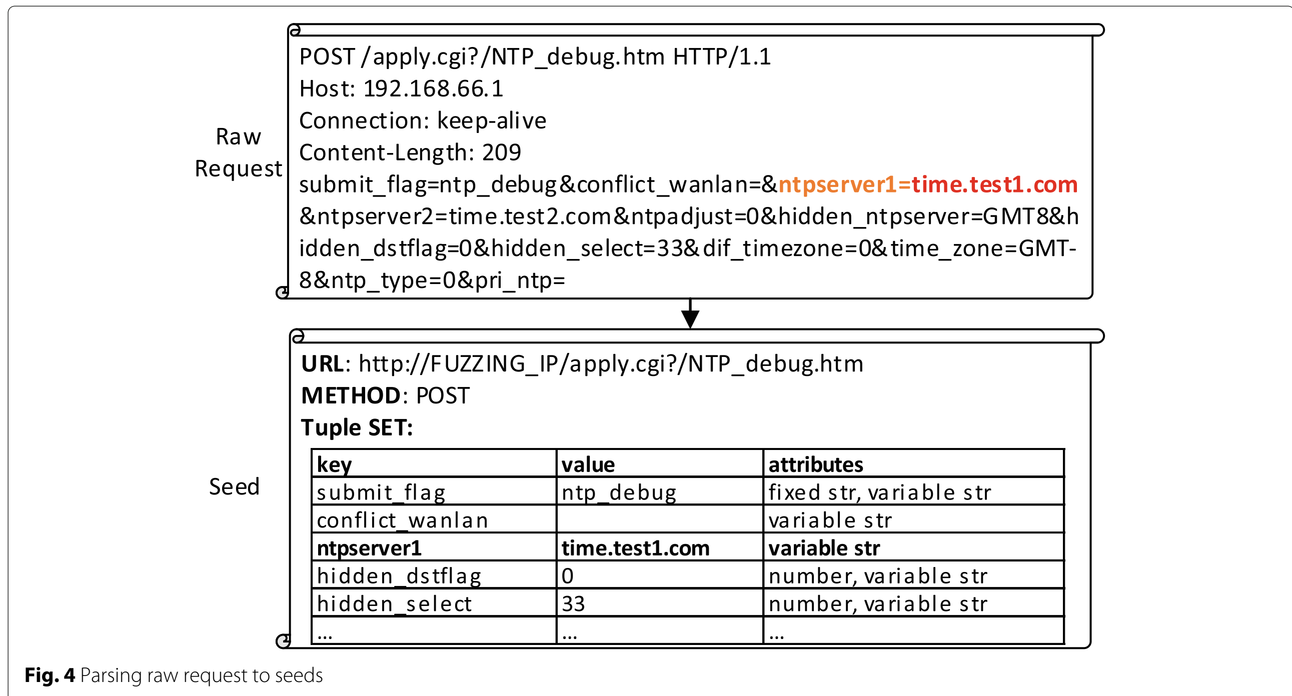


Fig. 4 Parsing raw request to seeds

mutation of the value of a k-v pair. As a result, after parsing k-v pairs from the raw requests and labeling them, the seeds can be generated more effectively.

CONF-READ Communication Model. When the user access frontend to get the current NTP configuration, the READ operation will call the `read_ntpserver1()`, a READ handler in the backend, to get the corresponding configuration from the device database.

A *stack-based overflow* issue is shown in Line 12 of Fig. 3. This issue exists in function `read_ntpserver1()`, which does not check the length of `ntp` before using it. To notice that, the value of `ntp` is set in Line 3 of function `conf_ntpserver1()`, and its length constraint is inconsistent with the length of `info` variable. Therefore, the vulnerability could be triggered when reading a malformed domain name from the database, which is set in Line 7, with length between 0x50 and 0x80.

The last vulnerability is a *cross-site scripting*. This issue exists in Line 12 and 13. We assue that the above stack-based overflow is not triggerred in Line 12. And the variable `info` get the value of `ntpserver`, then it is passed to the function `show_info_in_frontend()`. Because there are no filter or encode protection mechanism for XSS issues, the `info` variable can cause a stored XSS issue.

In such a situation, a single request is not enough to trigger the vulnerability, so we design the CONF-READ communication model to form multiple-requests test input. This model consists of two related operations, i.e., CONF operation and READ operation. In CONF operation, the requests of setting or modifying the configurations of devices are constructed, while in READ operation, the requests of getting the corresponding configurations are also constructed.

In general, making CONF operation through the HTTP request is the most common way to modify the configuration of devices, so the vulnerability triggered in the READ operation relies on the proper CONF operation. However, there are several extra methods to execute the CONF operation to modify arbitrary k-v pairs. In this case, if we use the D-CONF operation which executes the `/bin/config` command directly to set the ntpserver k-v pair, the stack-based overflow issue and XSS issue above can be triggered more easily without depending on the previous normal CONF operation which calls the `conf_ntpserver1()` handler. Figure 5 shows a D-CONF operation example through the root shell of a device to configure the device database. By calling the `/bin/config` program, Line 7 configures the value of variable `ntpserver` with the value that can trigger this XSS issue.

```
1 / # /bin/config --help
2 Usage:
3     config show
4     config get name
5     config set name=value
6     config unset name
7 / # /bin/config set ntpserver="<script>alert(\"xss\")</script>"
8 / # /bin/config get ntpserver
9 ntpserver="<script>alert(\"xss\")</script>"
```

Fig. 5 Code Snippet of D-CONF Operation

https://t.me/learningnets

Lessons learned. In light of the analysis of the configuration workflow in Fig. 2 and the motivating example, we conclude the root cause of the vulnerability as data inconsistency. The value inconsistent with key `ntpserver1` causes the *command injection* which is triggered at CONF operation and the *cross-site scripting* which is triggered as READ operation. Meanwhile, the length inconsistent between `conf_ntpserver1()` and `read_ntpserver1()` causes the *memory corruption* which is triggered at READ operation.

In the C-R model, we can see there are two major methods to complete a CONF operation. One usually uses HTTP requests to set the k-v pairs in a special web page and it is also the most common method to configure the device. However, the other one can arbitrarily modify all configurations. For easy distinction, we call the first method as normal CONF operation and the other one as Direct CONF operation. With normal CONF operation, ESRFuzzer can fuzz the device in a general mode and in D-CONF mode with D-CONF operation.

By coupling the K-V model with the C-R model, ESRFuzzer could reveal data inconsistencies in both k-v paired data and temporally related requests. Therefore, it is capable of detecting deep bugs in the web server of the SOHO router.

Discovering multi-type vulnerabilities

Discovering and triggering multi-type vulnerabilities at either CONF or READ operation, is necessary yet difficult in this fuzzing framework. As aforementioned in "Fuzzing in depth" section, we can design various of mutation rules according to the K-V model. Especially for the value with "variable string" attribute, we design different mutation rules to trigger exceptional behaviors of overflow, NULL-pointer dereference, command injection, format string and stored XSS respectively. In addition, we design two modes of fuzzing to fit for the different scenerio: general mode fuzzing and D-CONF mode fuzzing. General mode fuzzing aims at both the issues of CONF and READ operation, and we establish different types of communications to trigger the vulnerabilities which occur in either CONF or READ operation. D-CONF mode fuzzing focuses on the READ operation issues for the devices that support this mode fuzzing.

Once a vulnerability is triggered in a device, there will be some exceptional behaviors such as a backend crash, an abnormal response or executing an unexpected command. To monitor the vulnerability in the backend, it is insufficient to use liveness check (Muench et al. 2018), a most common method for monitoring dedicated devices. The method monitors the exceptional behaviors by only checking the connection state. In fact, the normal connection state (such as status code 200 in HTTP) does not always indicate the normal behavior, e.g., triggering an injected command execution can also return the correct connection state.

We design two general monitoring mechanisms to catch the exceptional behaviors, i.e., a *response-based monitor* and a *proxy-based monitor*. The response-base monitor checks not only connection states but also response contents that might include extra information. The proxy-based monitor receives the network accesses from the target router. Inspired by the conclusion of (Chen et al. 2018), we also design an optional *signal-based monitor*. It could catch more memory corruptions (i.e., silent memory corruptions (Muench et al. 2018)) at the cost of implanting a compiled executable into the device. By coupling mutation rules with monitoring mechanisms, we can detect multi-type vulnerabilities.

Detailed design

In this section, we present the detailed design of ESRFuzzer. As shown in Fig. 6, ESRFuzzer consists of six modules to work coordinately. Once connecting to the router, it collects the valid seeds by the *seed generator* module and *Config generator*. Then it feeds the seeds into the *mutator* module to generate mutated requests based on various mutation rules. Finally, ESRFuzzer triggers and monitors the exceptional behaviors by the collaboration of the *mutator*, the *monitor* and the *power control* module. The orange module and submodule are used for the general mode fuzzing while the blue modules are used for the D-CONF mode fuzzing.

1. **Seed Generator.** To generate the initial test cases, i.e., seeds, for mutation and CONF operation, the Seed Generator collects raw requests by the Request Collector submodule and parses them into k-v pairs. Then it labels all k-v pairs with attributes, which could guide mutation later. The Request Collector submodule is composed of two parts, a general crawler as the default setting to collect requests automatically, and an optional passive crawler to collect more requests by interacting with users.
2. **Config Generator.** Seed Generator is a way to generate the seed of general mode fuzzing, however, there are several ways to execute D-CONF operation such as NVRAM configuration mechanism. NVRAM is widely used in the embedded device to store the device configuration and it consists of various k-v pairs (Chen et al. 2016). NVRAM configuration can be configured in several ways: "config" or "nvram" series program in device, "backup and restore configuration" feature in web, etc. With this mechanism, the D-CONF operation can be operated effectively. For devices that use this mechanism, Config Generator can obtain all k-v pairs configs and store them into the database.

https://t.me/learningnets

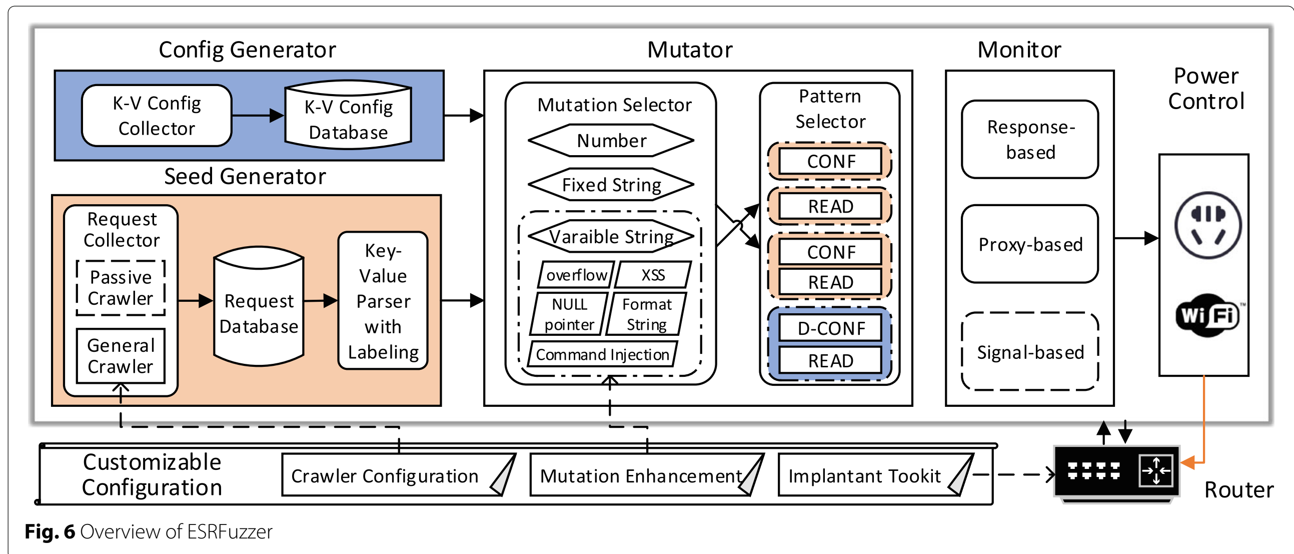


Fig. 6 Overview of ESRFuzzer

3. **Mutator.** The Mutator generates mutated requests and triggers vulnerabilities for the SOHO router through the cooperation of two submodules, i.e., the Mutation Selector and the Pattern Selector. Guided by the K-V model, mutation rules are selected and applied to the value of each k-v pair, according to the attributes of the k-v pair and the vulnerability type being discovered. In the Pattern Selector submodule, the types and the sequence of requests are decided based on the C-R model. The Pattern Selector could generate the communication request for the traditional fuzzing mode. It can generate one single request with the type of either CONF operation or READ operation. While, it could also generate a sequence of requests, e.g., a READ operation after a CONF operation. In the meanwhile, for the D-CONF mode fuzzing, it could also generate the mutation configurations and commit them to devices as D-CONF operations. Then it generates the corresponding READ operation for triggering the vulnerability.
4. **Monitor.** In order to collaborate with the Mutator module tightly and to monitor more exceptional behaviors, the Monitor module consists of two common monitors, a response-based monitor and a proxy-based monitor. The response-based one could usually monitor three types of vulnerability, i.e., memory corruption, XSS and information disclosure. To notice that, for information disclosure vulnerability, it monitors the response of a target URL without enough access permission. The proxy-based monitor is used for command injection and XSS vulnerabilities. In addition, an optional signal-based monitor is also provided to catch deeper memory corruptions. It is developed based on ptrace syscall and could monitor the signal such as SIGSEGV and SIGABRT.
5. **Power Control.** For the purpose of fuzzing the physical router continuously, a Power Control module is introduced. It is supported by a smart plug to control the power of the device. This module is controlled by the Monitor module. If the backend service is stuck into a “zombie” state, i.e., no response, a control command would be sent to the plug, then the device would be restarted.
6. **Configuration.** In order to improve the fuzzing efficiency, we also provide custom configurations for individual modules. All these configurations are optional. We configure the IP address of the default portal for the general crawler. We also provide mutation enhancement techniques for values with variable string attribute to trigger more exceptional behaviors. In order to ease the deployment of the signal-based monitor, we develop an implanting toolkit for the routers. With this toolkit, we can place the signal-based monitor into the device automatically.

Seed generation

This module aims at generating the seed for the following general mode fuzzing. We use the Request Generator to collect the raw requests then store them into the Request Database. Finally, we leverage the Key-Value parser to parse the requests into k-v pairs with attribute labeling.

As we mentioned in “Fuzzing in depth” section, a typical CONF operation is the second step of a web communication, as shown in Fig. 4. The Request Collector submodule aims at repeating this step and captures raw requests

https://t.me/learningnets

by two crawlers. By default, ESRFuzzer uses a general crawler to collect the requests automatically. However, the collection effects can be improved by the passive crawler.

The general crawler uses the default URL as an input, then it fills the web page automatically by parsing the input elements of the web page. In the meanwhile, it identifies all URLs of the page and then fills them recursively like a traditional crawler. It also stores the requests into the request database. In case of interaction during crawling, such as providing specific information, ESRFuzzer randomly select data from a predefined database to continue crawling.

The passive crawler is a semi-automatic toolkit which opens a web page and waits for the user input to fill the web page. After submitting the configuration, the passive crawler stores the requests into the request database and prepares for the next web page. Such a crawler is usually used to generate seeds from web pages with user input, such as the login page.

In addition, to collect the raw requests as many as possible and to facilitate the later attribute labeling procedure, both crawlers fill the same web page ten times.

As aforementioned, the K-V model describes the management protocol in a fine-grained manner. However, we can dig more information from the k-v pairs for deeper fuzzing. The KEY-VALUE parser analyzes the raw requests and split them into k-v pairs. Meanwhile, it labels the attributes of all k-v pairs according to their values.

There are two features in the value handling procedure of the backend. Firstly, the value is usually handled as a variable string, and the backend parses the crucial information to build related configuration. Secondly, there are always some validity checks, such as to judge whether a value is a number or a fixed string. As a result, if a fixed string is mutated, the check cannot be passed and the code protected by this check becomes unreachable. Therefore, we label a k-v pair with three type attributes, i.e., number, fixed string, and variable string. The attributes of a k-v pair determine the mutating rules applied to the k-v pair. By default, all k-v pairs are labeled with an attribute “variable string”. Algorithm 1 shows the attribute labeling process for k-v pairs.

In summary, Seed Generator converts each unique raw request to several seeds. Each seed contains the URL and the set of data tuples, each of which contains a key, a value and attributes. Figure 4 shows the converting process from a raw request to the seeds.

Algorithm 1 Labeling Attribute Algorithm.

Input: A set of k-v pairs P, each pair p contains $p.key$ and $p.value$;
Output: A set of tuple T, each tuple t contains $t.key$, $t.value$ and $t.attributes$

```
1:  T ← ∅
2:  for all p in P do
3:      t ← p
4:      if t.value is a number then
5:          add_attribute(t.attributes, “number”)
6:      end if
7:      if t.value does not change in various raw requests
    then
8:          add_attribute(t.attributes, “fixed string”)
9:      end if
10:     add_attribute(t.attributes, “variable string”)
11:     T.append(t)
12: end for
        return T
```

Config generation

This module aims at generating the seed for the following D-CONF mode fuzzing. There are two steps to obtain the seed: (i) Getting all k-v pairs in the NVRAM through the shell command like “nvram show”. (ii) Getting the mapping of the k-v pair and corresponding backend handler.

Backing to the Fig. 3 in “Motivation” section. `read_ntpserver1()` get the value of NVRAM variable “ntpserver1” by calling the `get_config()` function. In this case, we know when the backend hanlder `read_ntpserver1()` is called, the “ntpserver1” will be loaded and the configuration will be processed. So if the “ntpserver1” NVRAM variable is configured by the D-CONF operation, ESRFuzzer can craft a READ operation that calls the `read_ntpserver1()` function to trigger the overflow and cross-site scripting issue.

To understand backend handler and corresponding k-v pairs, ESRFuzzer will analyze the backend program which can be obtained by binwalk (devttys0 2013). There are two typical features to help the analysis. Firstly, backend handlers are always stored in a function pointer array in the data section of the backend program. And they are always called indirectly. So we can linear scan the data section of a program and get all backend handlers. Secondly, as Chen (Chen et al. 2016) said, the backend program usually call the NVRAM-related functions (`nvram_get()`, `nvram_set()`, etc.) by using a shared library named `libnvram.so`. What's more, the NVRAM-related functions use the k-v pair which is the same as the NVRAM variable as their arguments. So we can know what k-v pairs are loaded by a read handler through scanning the reference of NVRAM-related functions. For different vendors or devices, there could be a minor difference between the library or function name, but it is not important. Algorithm 2 shows the D-CONF seed generation algorithm.

https://t.me/learningnets

Algorithm 2 Algorithm for D-CONF Seed Generation.

Input: The backend program *BP*, all set of k-v pairs *P*;
Output: A set of D-CONF seed *DSeed*, each element *dseed* contains a backend handler *handler* and a set of k-v pairs P_h;

```
1:  DSeed ← ∅
2:  H ← parse_handlers_from_binary(BP)
3:  for all handler in H do
4:      tmp_set ← ∅
5:      P_k ← parse_keys_from_handler(handler)
6:      for all k in P_k do
7:          if has_this_key(P, k) then
8:              tmp_set.append(k, P[k])
9:          else
10:             tmp_set.append(k, "")     ▷ Set empty value for the key that is not in P
11:         end if
12:     end for
13:     dseed ← (handler, tmp_set)
14:     DSeed.append(dseed)
15: end for
        return DSeed.
```

Algorithm 3 Mutation Algorithm for each Seed.

Input: A single seed *S* which contains the *URL* and the set of tuple, T_n; Mutation option *option*;
Output: A mutated seed S_m;
Require: The set of mutation rules for variable strings, R_k;

```
1:  S_m ← S
2:  if option is "TUPLE" then
3:      T_r ← select_from_set(T_n, random(n)).
4:      for all t in T_r do
5:          attr ← select_from_set(t.attributes, 1)
6:          if attr is "number" then
7:              t.value ← mutate_number(t.value).
8:          end if
9:          if attr is "variable string" then
10:             rule ← select_from_set(R_k, 1)
11:             t.value ← mutate(t.value, rule).
12:         end if ▷ Do nothing for "fixed string" attribute
13:         S_m.T_n[ t.key] ← t
14:     end for
15: else
16:     S_m.URL ← mutate_URL(URL)
17: end if
        return S_m.
```

Mutation

From "Fuzzing in depth" section, we know the most crucial factor to trigger the vulnerability is the mutated value. There are two guidances to build mutation rules. Firstly, the root cause of the vulnerabilities is data inconsistency, especially for the variable string. So how to mutate the value of each k-v tuple is more important. Secondly, there are obvious differences between different types of vulnerability, so mutation rules should trigger exceptional behaviors according to the type of vulnerability.

Algorithm 3 shows the mutation algorithm for each seed. We separate the mutation of tuples and URL because it is inefficient to mutate them together. To mutate seeds, we select random number of tuples from each seed. According to the attributes of each tuple, we mutate its value with a related mutation rule. The mutation rules for number and fixed string attributes are simple, while for variable string, there are five mutation rules to trigger different types of exception behavior:

1. **For Overflow:** To trigger the overflow vulnerability, this framework usually duplicates the original value several times. If the key with an empty value, it assigns the key with a random number of payloads selected from a predefined database.
2. **For NULL-pointer dereference:** For the key with the non-empty value, this framework provides the empty value to trigger the potential NULL-pointer dereference vulnerability.
3. **For Command Injection:** To cooperate with the proxy-based monitor, ESRFuzzer constructs the value with the malformed payloads which are based on built-in tools such as `ping` or `wget`. If a command injection is triggered, these payloads will connect to the outside proxy-based monitor which contains a proxy server. If this monitor catches the requests sent from the router, it collects the detailed information about the exceptional behavior. It can also help to locate the vulnerability efficiently for the follow-up analysis.
4. **For Stored XSS:** There are two rules to construct the payloads for XSS. The most common payload contains the malformed JavaScript code to eject a message box. If the response-based monitor catches the message box that contains the string prefix with "xss_", it records the exception and locates the vulnerability tuple. The other payload constructed for the proxy-based monitor is similar to the rule for command injection, e.g., "$<$ *script* $>$(new Image()).src= "http://PROXY_\SERVER/MUTATION_KEY_INFO/" </script>".
5. **For Format String:** To trigger the format string vulnerability in a monitorable way, this framework usually concatenates the duplicated "%s" format string to the original value. Bacause in printf() family of functions, the "%s" type field will take the next argument of the stack and print it as a string. So if

https://t.me/learningnets

malformed duplicated “%s” format string is injected, the format string vulnerability will cause a crash.

For all mutation rules of variable string, we also use special characters, e.g., “;”, “$”, whitespace characters, and all kinds of quotation marks, to trigger more exceptional behaviors. These special characters can help to bypass the validity checks in the backend. For example, the `inet_addr(const char *cp)` function only extracts the part of a string before the first whitespace to check whether it is a valid IP address. Therefore, a value with the form “IP+Whitespace+Additional String” is wrongly considered as a valid IP address. For the 3rd and 4th mutation rules, we encode the key into the mutation value to assist in locating the exactly k-v pair.

For the URL mutation, this framework generates URLs containing special paths or sensitive file paths such as “/etc/passwd” or “/etc/shadow” beyond the permission when fuzzing. If a malformed URL can be accessed with a normal response, the response-based monitor will report it as an exceptional behavior.

Table 1 shows how these rules are applied to the motivating example illustrated in Fig. 4. The original values are “http://DEVICE_IP/apply.cgi?/NTP_debug.htm” and “time.test1.com”.

Triggering the exceptional behavior

For the general mode fuzzing, the common fuzzing method is to monitor the response status after sending a mutation packet. There is always one communication in this procedure. However, it is only useful for vulnerabilities that occur in CONF operation of the C-R model. We also need to trigger the vulnerabilities that occur in READ operation. To overcome this limitation, we trigger a READ operation after a CONF operation immediately. We also separate the request phases and monitoring methods. Moreover, we rollback CONF operation with the original value after each communication cycle.

Table 1 Example of Mutation Rules for Variable Strings

Section	Mutation Rule	Example of Mutated Value
ntpserver1	Overflow	time.test1.comtime.test1.com... (repeat 20 times)
	NULL-pointer dereference	(empty value)
	Command Injection	time.test1.com";wget http://PROXY_SERVER/ntpserver1;
	Cross-site scripting	time.test1.com";< *script* >alert('xss_ntpserver1')</script>
	Format String	time.test1.com%s%s%s%s%s%s%s%s%s%s
URL	http://DEVICE_IP/apply.cgi?/NTP_debug.htm/../../etc/passwd	

For the D-CONF mode fuzzing, the D-CONF operation can be executed directly by “nvram set” series command. In this way, the command shell of the device is required. As aforementioned, the D-CONF operation can not trigger any issues without a corresponding READ operation. According to the mapping of device handlers and NVRAM variables obtained by Algorithm 2, ESR-Fuzzer can craft a proper READ operation to trigger issues.

Monitoring the exceptional behavior

As we mentioned in “Fuzzing in depth” section, because “liveness check” is limited to the valid information from the response, it can hardly monitor various types of vulnerability nor catch deeper exceptional behaviors such as memory corruptions that occur in the subprocess. To make up for its limitation, we expand the liveness check into a response-based monitor. Moreover, we design the proxy-based monitor and the signal-based monitor to improve the monitoring performance in depth. Figure 7 shows three typical monitoring mechanisms with their monitoring scopes.

1. **Response-based monitor:** Besides *liveness check*, analyzing the response content of the communication can monitor XSS issues that triggered in the frontend. We craft the payload for each key to mutate during CONF operation and analyze the response content during the separated READ operation. Then we can locate the crucial keys that trigger the exceptional behaviors easily. Moreover, information disclosure can also be monitored by judging the response status.
2. **Proxy-based monitor:** For command injection and XSS issues, the Mutator module crafts malformed payloads to request the server of router outside. We build a monitoring server in the local network where the router can approach. So we can detect command injection and XSS issues when the monitoring server is accessed by the router. By cooperating with the crafted payload, the proxy-based monitor can efficiently locate the vulnerable URL and the crucial k-v pairs that trigger the exceptional behaviors.
3. **Signal-based monitor:** For the memory corruptions, the most common signals are SIGSEGV and SIGABRT. These two signals always occur when a process even though a subprocess crashes. So monitoring these signals can catch more true positive exceptional behaviors with less false positives. For Linux-based routers, we can develop the monitor based on `ptrace` syscall.
   Although the signal-based monitor can monitor the memory corruption more widely and accurately than the response-based one, it requires permission to implant a binary into the router, which is not always

https://t.me/learningnets

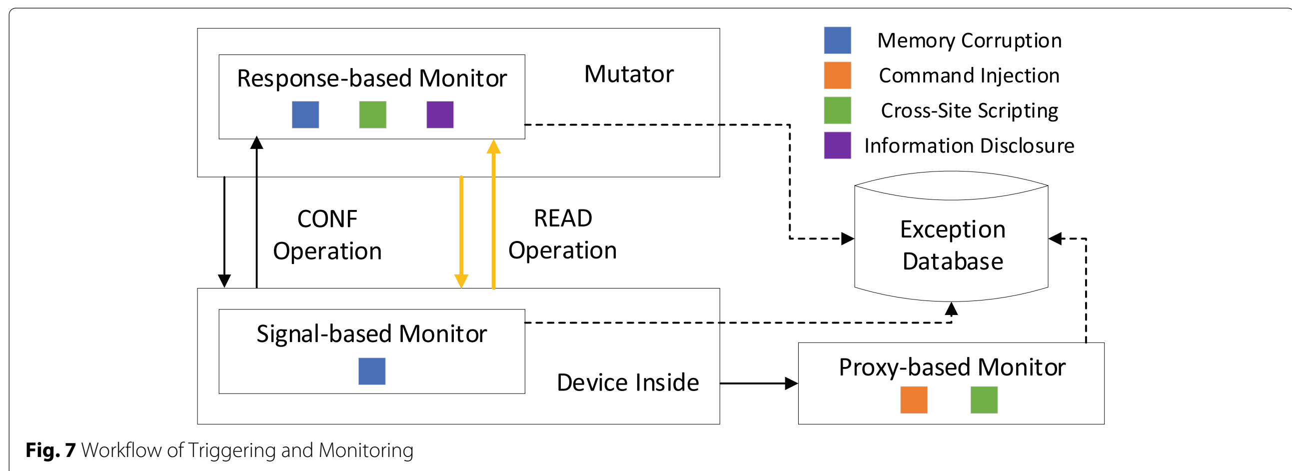


Fig. 7 Workflow of Triggering and Monitoring

> the case. So only the devices that satisfy the permission requirement can take advantage of this monitor. There are three ways to acquire this permission, i.e., through debug shell, exploiting known vulnerability such as command injection, and connecting the built-in serial port. We also develop an implant toolkit to put an executable into the device automatically.

Because three monitors work individually, we have to synchronize requests and exceptional behaviors for them. We record a timestamp for each request and a timestamp for each exceptional behavior, to make sure the request related to an exceptional behavior.

Experiment and evaluation

Implementation

We have implemented the automatic fuzzing framework modularly. For the Seed Generator module, we implemented a general crawler and a passive crawler both with Selenium (Selenium 2004). For the Config Generator module, we use IDA python script to generate the D-CONF seed. For the Monitor module, we implemented the three types of monitors independently. Specifically, the signal-based monitor is implemented in C with `strace` (strace 2000) and is implanted into devices by the help of device feature, known vulnerabilities or the serial port if possible. It is cross-compiled with Buildroot (buildroot 2001) as a statically linked binary. Now, it supports multiple architectures, including x86, x86-64, ARM32 (LE), ARM32 (BE), MIPS32 (BE) and MIPS32 (LE).

In order to restart the device when it hangs, ESRFuzzer firstly monitors the device status by trying to establish the TCP connections to the target device repeatedly. Then it restarts the device by the Power Control module if the TCP connection could not be established normally several times. We implemented this module based on the Mi Smart Plug (Mi Smart Plug 2015) with the help of python-miio (rytilahti 2018) protocol.

Experiment setting

In the experiment, we selected 10 devices from five different vendors to test. All of them support the general mode fuzzing, while 7 of them support the D-CONF mode fuzzing. Table 2 shows their information. The 5th column of the table shows the methods, to acquire permission to implant signal-based monitor, supported by each device. The last column shows the D-CONF mode fuzzing support of each device.

Except signal-based monitor, all modules of ESRFuzzer were deployed on Ubuntu 16.04. It connected the router by cable and a Mi Smart plug via a stable wireless connection. Figure 8 shows the topology of the experiment network. ESRFuzzer fuzzed each device continuously for 40 hours with general mode fuzzing and 20 hours with D-CONF mode fuzzing because the convenient D-CONF operation executed more quickly. During the fuzzing process, it would restart the device if the Power Control module does not receive any response for more than six minutes.

Overall experiment result

In total, as Table 3 shows, we confirmed total 136 unique issues for general mode fuzzing (101) and D-CONF mode fuzzing (35). The number of seeds that collected for general mode Fuzzing and D-CONF mode fuzzing are given in columns 2-3. What' more, we manually complete all of the PoCs. For memory corruption, the PoC can cause the backend crash or hijack its control flow. For command injection, the PoC can execute a shell command such as "reboot". For XSS, the PoC can eject a message box in the browser with the content "Hello, XSS". For information disclosure, the PoC can disclose some sensitive information.

https://t.me/learningnets

Table 2 Information of Routers under Fuzzing

ID	VENDOR	PRODUCT	ARCHITECTURE	SIGNAL-BASED MONITOR	D-CONF SUPPORT
1	NETGEAR	Orbi	ARM32 (LE)	Device Feature, Serial Port	Support
2	NETGEAR	Insight*	ARM32 (LE)	Not Support	Not Support
3	NETGEAR	WNDR-4500v3	MIPS32 (BE)	Device Feature, Serial Port	Support
4	NETGEAR	R8500	ARM32 (LE)	Device Feature, Serial Port	Support
5	NETGEAR	R7800	ARM32 (LE)	Device Feature, Serial Port	Support
6	TP-Link	TL-WVR900G	MIPS32 (BE)	Not Support	Not Support
7	Mercury	Mer450	MIPS32 (BE)	Not Support	Not Support
8	Tenda	G3	ARM32 (LE)	Existed Vulnerability	Support
9	Tenda	AC9	ARM32 (LE)	Existed Vulnerability	Support
10	Asus	RT-AC1200	MIPS32 (LE)	Device Feature	Support

*__Insight__ is short for "Insight Managed Smart Cloud Wireless Access Point"

After reported to the related vendors under the responsible disclosure policy, 120 of 136 unique issues have been confirmed by their vendors as our discovery, 11 of the rest 16 issues are under assessing process , 1 issue has been reported by others and the vendor considered the rest 4 issues as device design and won't fix them. These 120 vulnerabilities are assigned official IDs, including 43 CVE IDs[5], 73 PSV IDs[6] and 4 CNVD IDs.

Furthermore, we evaluated the impact of 120 officially confirmed vulnerabilities from their CVSS version 3 scores (NVD 2015) and their effects, which are shown in Fig. 9. We use the official four rankings based on CVSS scores to show the severity of the vulnerabilities. Specifically, more than one-fifth of issues (25/120) are at a high severity level. Vulnerabilities newly founded in D-CONF mode fuzzing are almost all at a medium severity level. We also counted the number of issues for four categories according to their effects, i.e., escalation of privilege (EoP), scripts execution, denial of service (DoS), and information disclosure. The majority of issues fall into EoP category, since we can craft their PoCs to hijack the control flow to execute a command of the low-level system, and hence we escalate the web-management privilege to the root privilege. Almost all PoCs can compromise the target device with one single message.

Besides, the D-CONF mode fuzzing aims at detecting the READ-op issues, and the shell permission is a convenient way to automatic the fuzzing procedure. Although the security levels are almost READ-op issues all medium priority especially found by D-CONF fuzzing, these issues are also harmful in some scenarios. For example, for a device that does not support a debug shell or we have no existing vulnerabilities, we can find a READ-op issue through D-CONF mode fuzzing with a device shell obtained by a serial port. Thus, we can obtain the root shell with this issue with restore configuration (CONF operation) and related READ operation. Actually, this method is significantly different from by serial port because it neither needs to disassemble a real device nor welds the pins. Attackers could escalate the privilege and obtain the root shell remotely if they obtain permission to trigger the READ-op issue. This issue can improve the security priority of the whole cyber-attack chain, which is also the reason that vendors treat them as issues with medium priority.

[5]We haven't disclosed all of the CVE IDs on the *oss-security mailing list* after being assigned, so not all of them can be found through the Internet now.
[6]The vendor hasn't listed all vulnerabilities on its security advisory yet, which causes several of the PSV IDs cannot be found through the vendor security advisory now.

Confirmed issues

General mode fuzzing can test the SOHO router in a general way for both issues that occurred in CONF operation or READ operation. During the general mode fuzzing, we totally confirmed 101 unique issues and 97 of these issues have been confirmed by their vendors, including 43 CVE IDs, 52 PSV IDs and 2 CNVD IDs. Their details are shown in Table 4.

For each device, the number of issues for each aforementioned type (i.e., MEM for memory corruptions, CMD for command injections, XSS for cross-site scripting, and INFO for information disclosure) grouped by two triggering phases (i.e., CONF and READ) is given in columns 2–7. Among the 101 confirmed issues, 48 issues are memory corruptions (47.52%) while the other includes 39 command injection issues (38.61%), 9 XSS issues (8.91%) and 5 information disclosure issue (4.95%). There are 67.33% of issues triggered in CONF operation and 32.67% in READ operation. All of the five devices that are confirmed more than ten issues were discovered with multiple types of vulnerabilities, except for TL-WVR900G. After analyzing its implementation, we have found that its backend is implemented by Lua language, which could avoid memory corruption.

https://t.me/learningnets

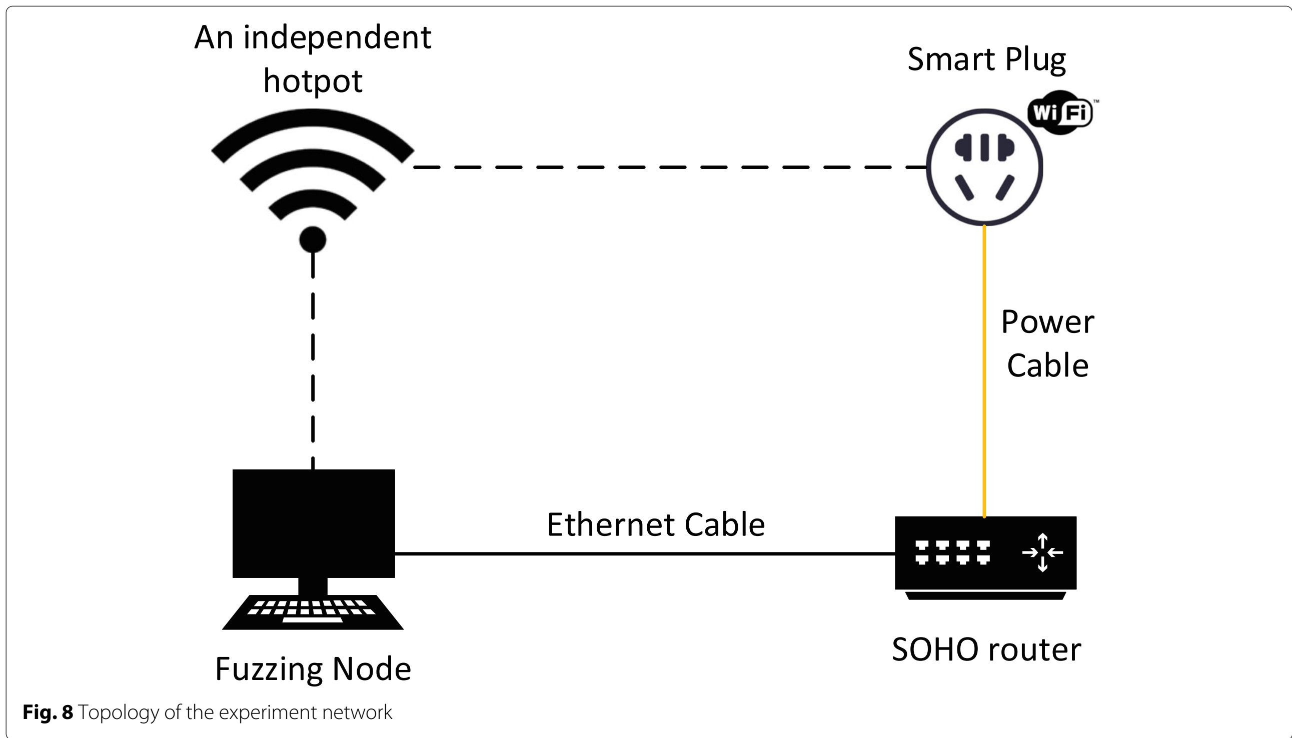


Fig. 8 Topology of the experiment network

Table 3 Overall Confirmed Issues of Fuzzing

Product	General Mode Seed	D-CONF Mode Seed	General Mode Issues	D-Conf Mode Issues	Total Issues
Orbi	216	314	2	9	11
Insight	108	N/A	1	N/A	1
WNDR-4500v3	188	266	16	0	16
R8500	208	503	13	4	17
R7800	232	295	24	8	32
TL-WVR900G	176	N/A	25	N/A	25
Mer450	91	N/A	2	N/A	2
G3	98	92	5	4	9
AC9	111	83	12	5	17
RT-AC1200	168	225	1	5	6
SUM	1596	1778	101	35	136

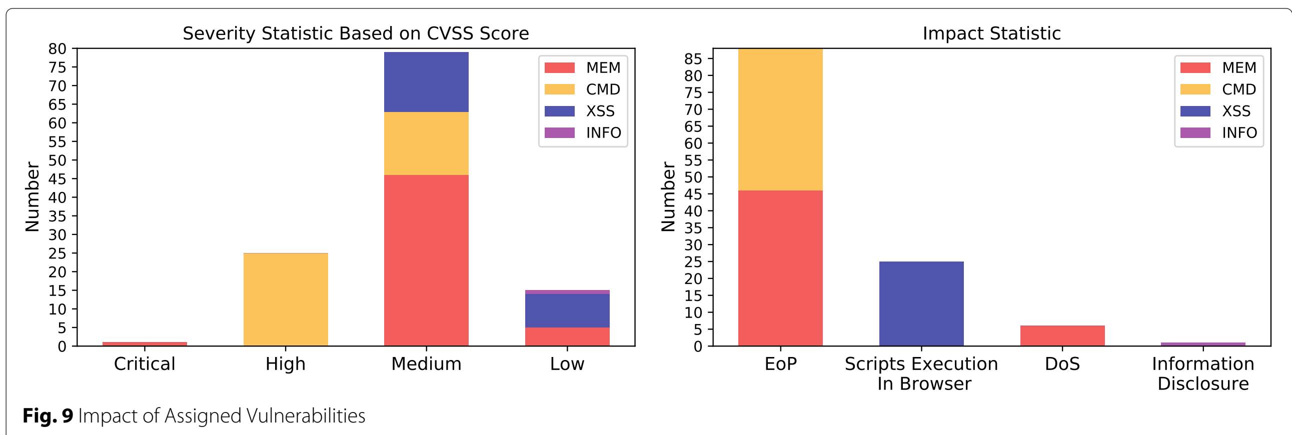


Fig. 9 Impact of Assigned Vulnerabilities

https://t.me/learningnets

Table 4 Confirmed Issues and their Types of General Mode Fuzzing

PRODUCT	CONF		READ				SUM
	MEM	CMD	MEM	CMD	XSS	INFO	
Orbi	0	0	0	0	1	1	2
Insight	0	1	0	0	0	0	1
WNDR-4500v3	6	2	7	0	0	1	16
R8500	9	0	0	0	3	1	13
R7800	0	8	10	0	5	1	24
TL-WVR900G	0	24	0	1	0	0	25
Mer450	0	2	0	0	0	0	2
G3	5	0	0	0	0	0	5
AC9	11	0	0	1	0	0	12
RT-AC1200	0	0	0	0	0	1	1
SUM	31	37	17	2	9	5	101

D-CONF mode fuzzing can execute the CONF operation without any backend check that exists in general mode fuzzing. So it can find some new issues easily. During the D-CONF mode fuzzing procedure, we totally confirmed 35 new unique issues. After being reported to the related vendors under the responsible disclosure policy, 23 of 35 issues have been confirmed by their vendors and assigned official IDs, including 21 PSV IDs and 2 CNVD IDs. What's more, 1 command injection issue has been reported by others. Details of these confirmed issues are shown in Table 5.

Among the 35 confirmed issues, 12 issues are memory corruption (34.29%), 4 issues are command injection (11.43%) and 19 issues are XSS (54.29%). Comparing this result with general mode fuzzing, there are two obvious conclusions:

1. **Command injection issues less likely occur in READ operation.** The command injection are both the least issue in READ operation issues discovered by general and D-CONF mode fuzzing. This shows that the command injection issue usually occurs in CONF operation for the reason that the configuration operation is always executed during the procedure of CONF operation.
2. **D-CONF mode fuzzing can trigger XSS issues more easily.** In D-CONF mode fuzzing, XSS issues account for 54.29% of the whole confirmed issues, while it only accounts for 27.27% of the confirmed READ operation issues in general mode issues. Comparing to other types of vulnerability, the payload of XSS contains more special characters and it is more complex. So in general mode fuzzing, the CONF operation are more likely to filter the payload and the value cannot be set successfully. D-CONF operation removes this obstacle so XSS issues can be triggered more easily.

Table 5 Confirmed Issues and Their Types of D-CONF Mode Fuzzing

PRODUCT	READ			SUM
	MEM	CMD	XSS	
Orbi	0	0	9	9
WNDR-4500v3	0	0	0	0
R8500	2	0	2	4
R7800	0	3	5	8
G3	4	0	0	4
AC9	4	1	0	5
RT-AC1200	2	0	3	5
SUM	12	4	19	35

Effect of monitors

The distribution of different confirmed issues caught by different monitors of General Mode fuzzing are shown in Table 6. We can observe that most of the confirmed issues (77.23%) are caught by the response-based monitor and the proxy-based monitor, showing the effectiveness of these device-independent monitors. We also find additional 23 issues are caught by the signal-based monitor, showing its ability to discover deep memory corruption vulnerabilities. To notice that, signal-based monitor can catch much more issues for WNDR-4500v3 and R7800. It is caused by their special implementation. In their backends, they create subprocesses to handle the requests and always respond with "configuration failure" when subprocesses are crashed. In such cases, the signal-based monitor other than the response-based monitor can deal with them.

Table 7 shows the effectiveness of monitors for D-CONF mode fuzzing. In this mode, the response-based monitor works similarly with the signal-based monitor on memory corruption issues (10 vs. 12) and with proxy-based monitor on XSS issues (15 vs. 19).

For memory corruption issues, none of the devices that found issues handles the requests with subprocess, so the exceptional behaviors can be easily monitored by the response-based monitor.

For XSS issues, the payload for proxy-based monitor is more complex than for the response-based monitor. So in general mode fuzzing, the CONF operation with these payloads are more impossible to execute successfully than the payload for the response-based monitor. However, D-CONF operation makes this more easily.

Reboot

The times of reboot for each device during the fuzzing is given in Fig. 10. During 40 hours in our testing, each device rebooted 6.8 times on average. It is interesting that

https://t.me/learningnets

Table 6 Effectiveness of Monitors for 101 Confirmed Issues of General Mode Fuzzing

Product	MEM		CMD	XSS		INFO
	R	S	P	P	R	R
Orbi	0	0	0	1	0	1
Insight	0	N/A	1	0	0	0
WNDR-4500v3	3	10	2	0	0	1
R8500	7	2	0	1	2	1
R7800	2	8	8	2	3	1
TL-WVR900G	0	N/A	25	0	0	0
Mer450	0	N/A	2	0	0	0
G3	4	1	0	0	0	0
AC9	9	2	1	0	0	0
RT-AC1200	0	0	0	0	0	1
SUM	25	23	39	4	5	5

1. **R** represents response-based monitor. **S** represents signal-based monitor. **P** represents proxy-based monitor. **N/A** represents this monitoring method is not supported by the device
2. The issues of different monitors for the one type of vulnerability could be overlapped, the union of them equals to the total confirmed issues

R8500 rebooted 3.6x as many as R7800 did. This is because R8500 handles all requests in only one process, while R7800 handles the requests by creating subprocesses. Moreover, the devices with Openwrt-based (Fainelli 2008) operating system (e.g., TL-WVR900G) are more stable than others.

The times of reboot for D-CONF mode fuzzing is given in Fig. 11. During 20 hours of testing, each device rebooted 4.7 times on average. The reboot times per hour of R8500 in D-CONF mode (0.2) is significantly less than in general mode (0.425) while the RT-AC1200 is on the contrary. This is because device reboot are more likely caused by the memory corruption issues. In fact, the most memory corruptions of R8500 are triggered in CONF operation while of RT-AC1200 are triggered in READ operation.

Table 7 Effectiveness of Monitors for 35 Confirmed Issues of D-CONF Mode Fuzzing

Product	MEM		CMD	XSS	
	R	S	P	P	R
Orbi	0	0	0	7	9
WNDR-4500v3	0	0	0	0	0
R8500	2	2	0	2	2
R7800	0	0	3	3	5
G3	3	4	0	0	0
AC9	3	4	1	0	0
RT-AC1200	2	2	0	3	3
SUM	10	12	4	15	19

1. **R** represents response-based monitor. **S** represents signal-based monitor. **P** represents proxy-based monitor
2. The issues of different monitors for the one type of vulnerability could be overlapped, the union of them equals to the total confirmed issues

We compared ESRFuzzer(general mode fuzzing) with three popular open-source fuzzers in terms of discovering vulnerabilities. For memory corruption vulnerabilities, we chose *boofuzz* (jtpereyda 2012), a fork and successor of famous protocol fuzzer *Sulley* (Fitblip 2012), as the comparative tool. For command injection vulnerabilities, we chose *Commix* (Stasinopoulos et al. 2015) instead of *boofuzz*, as *Commix* are better with more mutation rules and monitoring methods. For XSS vulnerabilities, we selected *wfuzz* (jtpereyda 2014), a popular web fuzzing tool supporting XSS detection. ESRFuzzer did not compare with others in information disclosure vulnerabilities, as most of the results (4 out of 5) are being assessed. We will perform the comparison in future work.

In the comparison, seven devices among four vendors were selected. On these devices, we ran all of the tools for 40 hours without interruption, which was the same as ESRFuzzer. We fed them with the raw requests, which were also the same as ESRFuzzer. To satisfy the special input requirement of boofuzz, we converted each seed with k-v pairs into the data representation of boofuzz[7]. As shown in Fig. 12, ESRFuzzer outperformed those three comparative fuzzers in all types of vulnerabilities. Specifically, it has found more memory corruption issues than boofuzz by 53.57% and more command injection issues than Commix by 25.81%. Meanwhile, ESRFuzzer found one more XSS issue than wfuzz. We analyze the results in details.

Memory Corruption. *Boofuzz* cannot find any vulnerabilities with its default data representation. We encoded our seeds, which consists of k-v pairs, in boofuzz's data representation. In such a way, boofuzz can mutate request content field effectively, and trigger vulnerabilities. However, due to its lack of multiple monitor methods, boofuzz could miss issues that occur in READ operation and in a subprocess.

Command Injection. The true positives of *Commix* are less than those of ESRFuzzer for two reasons. Firstly, among its many monitor methods, "time-related" injection monitoring technique is suitable for the devices. However, the technique relies on response time, which makes Commix miss some short-time exceptions. Secondly, Commix only monitors exceptions during CONF operation, which make it ignore the issues that occur in READ operation, such as issues in AC9 and TL-WVR900G.

Cross-site Scripting (XSS). ESRFuzzer found one more XSS issue than *wfuzz*. It is in R8500. It is missed by wfuzz because the input generated cannot bypass the backend

[7] An example of the conversion to the data representation of boofuzz is listed in the Appendix

https://t.me/learningnets

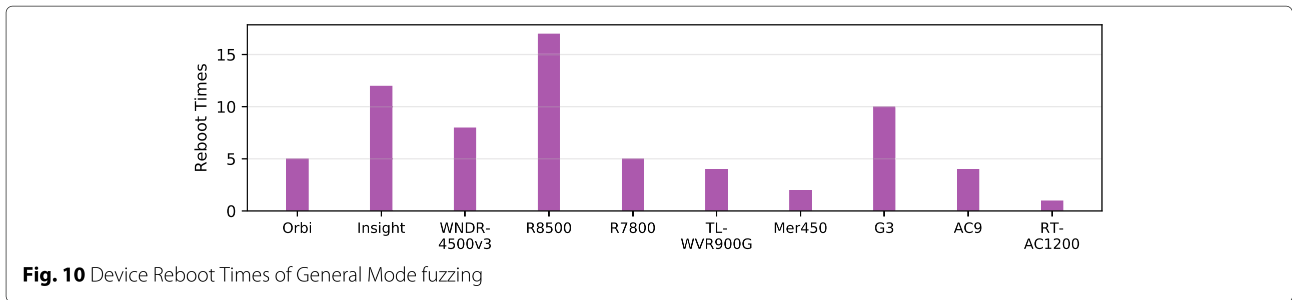


Fig. 10 Device Reboot Times of General Mode fuzzing

validity check without the guidance of K-V model. Therefore, the crafted value cannot be stored successfully through a CONF operation and the value returned to the frontend did not trigger a XSS issue.

Case study

In this section, a command injection vulnerability caused by the D-CONF operation will be introduced. This vulnerability has been first found by ISE Labs and was assigned with CVE-2020-15916 (NVD 2020) . Actually, it is a typical READ operation issue and we also found it during our D-CONF mode fuzzing.

```
TelnetTenda+0x9C
......
; Value of variable lan.ip is loaded
  into
; STACK_LAN_IP by GetValue() function.
01   SUB      R3 , R11,  #-STACK_LAN_IP
02   LDR      R2, =(LAN_IP - 0xFD53C) ;
   "lan.ip"
03   ADD      R2, R4, R2 ; "lan.ip"
04   MOV      R0, R2
05   MOV      R1, R3
06   BL       GetValue
07   LDR      R3, =(CMD_KILL_TELNETD -
  0xFD53C) ; "killall -9 telnetd"
08   ADD      R3, R4, R3 ; "killall -9
  telnetd"
09   MOV      R0, R3  ; command
10   BL       system
; Value of variable lan.ip is passed
  into
; doSystemCmd() function.
11   SUB      R3 , R11,  #-STACK_LAN_IP
12   LDR      R2, =(CMD_TELNET - 0
  xFD53C) ; "telnetd -b %s &"
13   ADD      R2, R4, R2 ; "telnetd -b
  %s &"
; the 1st arg is string "telnetd -b %s
  &"
14   MOV      R0, R2
; the 2nd arg is value of lan.ip
  variable
15   MOV      R1, R3
; doSystemCmd() function malforms a
  command
; with first and second arguments.
16   BL       doSystemCmd
17   LDR      R3, =(aOpDWlRateDInde - 0
  xFD53C) ; "op=%d,wl_rate=%d,index=1"
18   ADD      R3, R4, R3 ; "op=%d,
  wl_rate=%d,index=1"
......
```

Listing 1 Binary Code Snippet in ARM of TendaTelnet() Handler.

This issue is caused by the insufficient check of "lan.ip" key-value pair of TendaTelnet() READ handler. Listing 1 shows the binary code snippet of the vulnerable READ handler. In this code snippet, the variable "lan.ip" is loaded from the device database through the `GetValue()` function. Then its value is passed to the `doSystemCmd()` function which malformed a command with "telnetd -b %s &" and the value of "lan.ip"variable. In normal, the value of "lan.ip" should be a legal ip address such as "192.168.0.1". However, during the D-CONF mode fuzzing, a value "192.168.0.1; wget http://

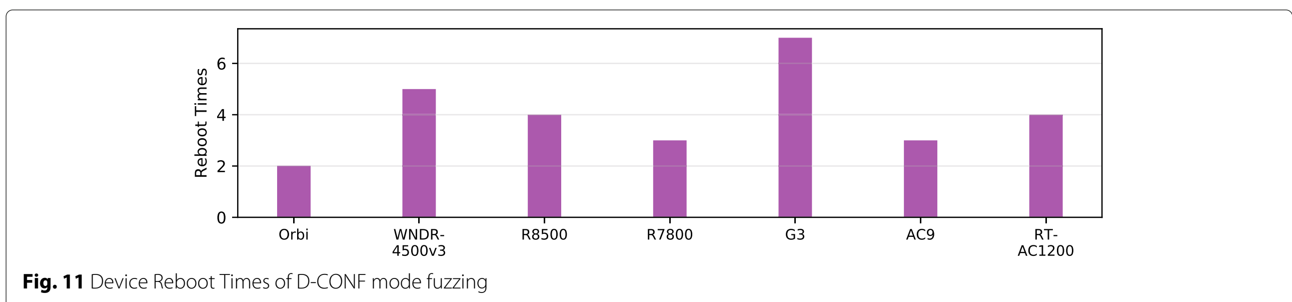


Fig. 11 Device Reboot Times of D-CONF mode fuzzing

https://t.me/learningnets

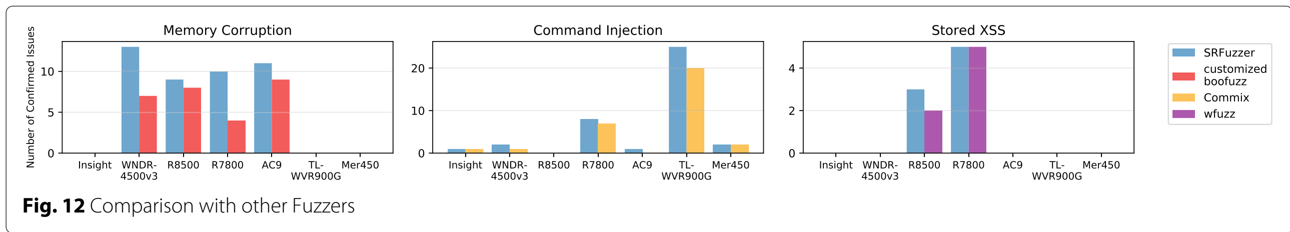


Fig. 12 Comparison with other Fuzzers

PROXY_SERVER_ADDRESS/lan.ip; " was set. Actually, after the D-CONF operation, this command injection has not been triggered. Only after the READ operation calling the TelnetTenda handler, this issue was triggered and our monitor caught it. In this case, ESRFuzzer malformed a GET request to "http://DEVICE_ADDRESS/goform/telnet" URL as the READ operation.

After analyzing this vulnerability manually, we found that the value of "lan.ip" variable could also be configured by a CONF handler that configures the LAN IP address. However, the vulnerable READ handler is used for enabling the build-in telnet service and is unrelated to this CONF handler. Therefore, General mode fuzzing cannot discover this issue because only the related READ handler could be triggered after a CONF operation. This issue also illustrates that Direct CONF mode fuzzing performs better than general mode fuzzing about the READ operation issue.

Discussion

In this section, we discuss the limitation of the current fuzzing framework and explore the improvement direction in the future.

Limitation of the Scope. The IoT device whose management protocols satisfy the C-R model and the K-V model can take advantage of ESRFuzzer. We will extend our work to apply more types of device such as camera, switch, and printer, as well as other widely used management protocols such as SOAP. Listing 2 show an SOAP request sample for NTP configuration, the KEY-VALUE pairs are emphasized with red and green.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://
    schemas.
xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header>
    <SessionID>ABCDEFGHIJKLMNOPQRST</SessionID>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
    <M1:SetNTP xmlns:M1="urn:ROUTER:service:
        DevConfig:
    1">
        <Option>Preferred</Option>
        <NTPServer1>time.test1.com</NTPServer1>
        <NTPServer2>time.test2.com</NTPServer2>
        <TimeZone>GMT-8</TimeZone>
    </M1:SetNTP>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 2 A SOAP Request Sample

Vulnerability of severity. Although ESRFuzzer can fuzz devices without authentication, it does not find any pre-authenticated issues in our experiment. So we will examine the attack surfaces more thoroughly, to find hidden interfaces, so as to enhance the ability to discover authentication bypass vulnerabilities.

Research on data inconsistency. Based on the C-R model and the K-V model, we notice that there are several data inconsistencies between different backend procedures for a specific k-v pair. In this paper, we focus on the automatic fuzzing process and leave the analysis of the semantic relevance systematically for future investigations. The in-depth research can also help vendors to consolidate their security design.

Monitoring. From the monitoring perspective, ESRFuzzer takes advantage of the two aspects: the various monitors for different types of vulnerability and the various monitoring mechanisms for the same type of vulnerability. We will try to improve the generality of the intrusive monitor, e.g., proposing a framework to repack the firmware or support direct flash writing.

Related work

There are many researches on detecting vulnerabilities of IoT devices, which might also be applied to SOHO routers. A. Costin et al. (Costin et al. 2014) performed a large scale analysis on the firmware images while not finding any issues at runtime. H. Bojinov et al. (Bojinov et al. 2009) audited several types of embedded management interface and B. Gourdin et al. (Gourdin et al. 2011) proposed *WebDroid* to build secure embedded web interfaces. Similar works (HP-Fortify-ShadowLabs 2014; Independent Security Evaluators 2017) for the SOHO devices are also proposed. Another type of large scale vulnerability detection is to scan for venerable devices in the internet. A. Cui (Cui and Stolfo 2010) presented a quantitative lower bound on the number of the vulnerable embedded devices on a global scale. They found over 540,000 embedded devices are configured with factory default root passwords.

Fuzzing is an effective method to automatically discover vulnerabilities. Feedback-driven fuzzers (CENSUS 2016; Google 2015a; Google 2015b; LLVM 2015; Zalewski 2014; Rawat et al. 2017) used the runtime code coverage to guide the following inputs generation. You (You et al. 2019),

https://t.me/learningnets

Jain (Jain et al. 2018) worked on the automatic input type inference of input bytes. Fuzzing process may stick at particular branches with complex conditions. To solve this problem, researchers combined the fuzzing with symbolic execution (Cha et al. 2012; Kim et al. 2017; Stephens et al. 2016). Though they are promising, they are not suitable for the routers because of lacking internal runtime information.

Emulating devices is a potential solution to get the runtime information. A. Costin et al. (Costin et al. 2016) used qemu to emulate the web interface of certain linux-based devices. The method cannot work for web interfaces that contain special hardware related operations, like Wi-Fi configuration. FIRMADYNE (Chen et al. 2016) emulated Linux-based COTS firmware by supporting the emulation of NVRAM of devices. But it was limited only for ARM-based devices in our practice.

To overcome the firmware acquisition and emulation problems, fuzzing on physical devices was proposed. Z. Wang et al. (Wang et al. 2013) developed RPFuzzer to fuzz the router protocol like SNMP. IoTFuzzer (Chen et al. 2018) was an app-based fuzzing framework which aimed at finding memory corruptions in physical IoT devices without firmware images. It took advantage of the collaboration of fuzzing and taint analysis. It only focused on mobile-to-web interface and detected memory corruptions with only liveness check, which was not enough for finding web server vulnerabilities. Moreover, M. Muench (Muench et al. 2018) analyzed the challenges of fuzzing embedded devices and presented six heuristics to detect memory corruptions.

Program analysis techniques are also used for IoT devices to discover vulnerability. Q. Feng et al. (Feng et al. 2016) adopted a graph-based method to search for vulnerabilities in firmware images. They converted control flow graphs to numeric feature vectors, and used several hashing techniques to achieve real-time search. Y. Shoshitaishvili et al. (Shoshitaishvili et al. 2015) used static symbolic execute and program slicing to find backdoor. Besides, dynamic symbolic execution were also used. FIE (Davidson et al. 2013) was a symbolic execution framework to find bugs for MSP430 firmware. Avatar (Zaddach et al. 2014) was a framework to coordinate emulator and device when analyzing the firmware. ESRFuzzer is a complement to these techniques.

Conclusion

We have presented ESRFuzzer to identify multi-type vulnerabilities of SOHO routers in a fully-automatic mode without device emulation. Based on the D-CONF operation which we found in popular SOHO routers, we extended the CONF-READ communication model and propose a novel method focusing on the detection of the READ-op issue by improving the SRFuzzer with D-CONF mode fuzzing mechanism. Finally, ESRFuzzer has discovered 35 previously unknown READ-op issues that belong to three vulnerability types, and 23 of them have been confirmed as 0-day vulnerabilities by vendors.

Appendix A: Data representation sample

Listing 3 shows a data representation sample for boofuzz, this sample is related to the requests from Fig.4.

```
s_static("submit_flag=")
s_string("ntp_debug")
s_static("&conflict_wanlan=")
s_string("")
s_static("&ntpserver1=")
s_string("time.test1.com")
s_static("&ntpserver2=")
s_string("time.test2.com")
s_static("&ntpadjust=")
s_string("0")
s_static("&hidden_ntpserver=")
s_string("GMT8")
s_static("&hidden_dstflag=")
s_string("0")
s_static("&hidden_select=")
s_string("33")
s_static("&dif_timezone=")
s_string("0")
s_static("&time_zone=")
s_string("GMT-8")
s_static("&ntp_type=")
s_string("0")
s_static("&pri_ntp=")
s_string("") s_block_end()
```

Listing 3 A Data Representation Sample for boofuzz

Appendix B: Assigned vulnerabilities

Table 8 shows the assigned CVE, PSV and CNVD ids during the fuzzing.

Table 8 All assigned vulnerability IDs

Model	Vulnerabity ID
NETGEAR Orbi	PSV-2017-3093 , PSV-2018-0554, PSV-2018-0555, PSV-2018-0556, PSV-2018-0557, PSV-2018-0558, PSV-2018-0559, PSV-2018-0560, PSV-2018-0561, PSV-2018-0562
NETGEAR Insight	PSV-2018-0610
NETGEAR WNDR-4500v3	PSV-2017-3169, PSV-2017-3167, PSV-2017-3170, PSV-2017-3168, PSV-2017-3154, PSV-2017-3158, PSV-2017-3159, PSV-2017-3152, PSV-2017-3165, PSV-2017-3166, PSV-2017-3157, PSV-2017-3156, PSV-2017-3160, PSV-2017-3155, PSV-2017-3153
NETGEAR R8500	PSV-2017-3065, PSV-2017-2460, PSV-2017-2427, PSV-2017-2428, PSV-2017-2254, PSV-2017-2226, PSV-2017-2229, PSV-2017-2228, PSV-2017-2227, PSV-2018-0244, PSV-2018-0243, PSV-2018-0242, PSV-2018-0614, PSV-2018-0618, PSV-2020-0255, PSV-2020-0261

https://t.me/learningnets

Table 8 All assigned vulnerability IDs (*Continued*)

Model	Vulnerabity ID
NETGEAR R7800	PSV-2018-0116, PSV-2018-0115, PSV-2018-0148, PSV-2018-0144, PSV-2018-0141, PSV-2018-0142, PSV-2018-0139, PSV-2018-0132, PSV-2018-0173, PSV-2018-0140, PSV-2018-0136, PSV-2018-0138, PSV-2018-0145, PSV-2018-0171, PSV-2018-0146, PSV-2018-0147, PSV-2018-0137, PSV-2018-0135, PSV-2018-0133, PSV-2018-0172, PSV-2018-0174, PSV-2018-0159, PSV-2018-0158, PSV-2018-0355, PSV-2018-0356, PSV-2018-0357, PSV-2018-0485, PSV-2018-0486, PSV-2018-0487, PSV-2018-0488, PSV-2018-0489
TP-Link TL-WVR900G	CVE-2017-15613, CVE-2017-15614, CVE-2017-15615, CVE-2017-15616, CVE-2017-15617, CVE-2017-15618, CVE-2017-15619, CVE-2017-15620, CVE-2017-15621, CVE-2017-15622, CVE-2017-15623, CVE-2017-15624, CVE-2017-15625, CVE-2017-15626, CVE-2017-15627, CVE-2017-15628, CVE-2017-15629, CVE-2017-15630, CVE-2017-15631, CVE-2017-15632, CVE-2017-15633, CVE-2017-15634, CVE-2017-15635, CVE-2017-15636, CVE-2017-15637
Mercury Mer450	CVE-2018-12488, CVE-2018-12489
Tenda G3	CVE-2018-12057, CVE-2018-12058, CVE-2018-12059, CVE-2018-12060, CVE-2018-12061
Tenda AC9	CVE-2018-8742, CVE-2018-8743, CVE-2018-8744, CVE-2018-8745, CVE-2018-8746, CVE-2018-8747, CVE-2018-8748, CVE-2018-8749, CVE-2018-8750, CVE-2018-8751, CNVD-2019-00015, CNVD-2019-00016
Asus RT-AC1200	CVE-2017-16901, CNVD-2020-58141, CNVD-2020-59431

Acknowledgements
Not applicable.

Authors' contributions
YZ, YYZ and WH designed this research. YZ, KPJ, JS, and LQL built this framework and performed experiments. YZ, YYZ, WH wrote this paper. YYZ, WH, CZ, BZL reviewed and edited the manuscript. All authors read and approved the manuscript.

Authors' information
Not applicable in this blinded manuscript version.

Funding
This work is supported in part by Chinese National Natural Science Foundation (61802394, U1836209, 62032010), National Key Research and Development Program of China (2016QY071405), Strategic Priority Research Program of the CAS (XDC02040100, XDC02030200, XDC02020200), Program No.2017-JCJQ-ZD-043-01, and BNRist Network and Software Security Research Program (BNR2019TD01004, BNR2019RC01-009).

Availability of data and materials
All public dataset sources are as described in the paper.

Declarations

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. [3]Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing, China. [4]Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China. [5]Institute for Network Science and Cyberspace, Tsinghua University, Beijing, China. [6]Beijing National Research Center for Information Science and Technology, Beijing, China.

Received: 7 January 2021 Accepted: 19 April 2021
Published online: 19 July 2021

References

ACI (2018) Securing IoT Devices: How Safe Is Your Wi-Fi Router? https://www.theamericanconsumer.org/wp-content/uploads/2018/09/FINAL-Wi-Fi-Router-Vulnerabilities.pdf. Accessed 1 May 2019
Bojinov H, Bursztein E, Lovett E, Boneh Dan (2009) Embedded management interfaces: Emerging massive insecurity, Las Vegas, Nevada
buildroot (2001) Buildroot - Making Embedded Linux Easy. https://buildroot.org/. Accessed 1 May 2019
CENSUS (2016) Choronzon - An evolutionary knowledge-based fuzzer. https://github.com/CENSUS/choronzon. Accessed 1 May 2019
CERT (2016) Multiple Netgear routers are vulnerable to arbitrary command injection. https://www.kb.cert.org/vuls/id/582384/. Accessed 1 May 2019
Cha SK, Avgerinos T, Rebert A, Brumley D (2012) Unleashing mayhem on binary code. In: IEEE Symposium on Security and Privacy (S&P). IEEE, San Francisco
Chen J, Diao W, Zhao Q, Zuo C, Lin Z, Wang X, Lau WC, Sun M, Yang R, Zhang K (2018) Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In: Network and Distributed System Security Symposium (NDSS). ISOC, San Diego
Chen DD, Egele M, Woo M, Brumley D (2016) Towards automated dynamic analysis for linux-based embedded firmware. In: Network and Distributed System Security Symposium (NDSS). ISOC, San Diego
Costin A, Zaddach J, Francillon A (2016) Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In: ACM Asia Conference on Computer and Communications Security (ASIACCS). ACM, Xi'an
Costin A, Zaddach J, Francillon A, Balzarotti D (2014) A large-scale analysis of the security of embedded firmwares. In: USENIX Security Symposium. USENIX Association, San Diego
Cui A, Stolfo SJ (2010) A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In: Annual Computer Security Applications Conference (ACSAC). IEEE, Orlando
Davidson D, Moench B, Ristenpart T, Jha S (2013) Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In: USENIX Security Symposium. USENIX Association, Washington, D.C.
devttys0 (2013) Binwalk: Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk. Accessed 1 May 2019
Fainelli F (2008) The openwrt embedded development framework. In: Free and Open Source Software Developers European Meeting (FOSDEM)
Feng Q, Zhou R, Xu C, Cheng Y, Testa B, Yin H (2016) Scalable graph-based bug search for firmware images. In: ACM Conference on Computer and Communications Security (CCS). ACM, Vienna
Fitblip (2012) Sulley - a pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley. Accessed 1 May 2019
Google (2015) Honggfuzz. https://github.com/google/honggfuzz. Accessed 1 May 2019
Google (2015) syzkaller - linux syscall fuzzer. https://github.com/google/syzkaller. Accessed 1 May 2019
Gourdin B, Soman C, Bojinov H, Bursztein E (2011) Toward secure embedded web interfaces. In: USENIX Security Symposium. USENIX Association, San Francisco
HP-Fortify-ShadowLabs (2014) Report: Internet of Things Research Study. https://www8.hp.com/us/en/hp-news/press-release.html?id=1744676. Accessed 1 May 2019
Independent Security Evaluators (2017) SOHO Network Equipment (Technical Report). https://www.securityevaluators.com/wp-content/uploads/2017/07/soho_techreport.pdf. Accessed 1 May 2019
Jain V, Rawat S, Giuffrida C, Bos H (2018) Tiff: Using input type inference to improve fuzzing. In: Annual Computer Security Applications Conference (ACSAC). IEEE, San Juan
jtpereyda (2012) A fork and successor of the Sulley Fuzzing Framework. https://github.com/jtpereyda/boofuzz. Accessed 1 May 2019
jtpereyda (2014) Wfuzz - The Web Fuzzer. https://github.com/xmendez/wfuzz. Accessed 1 May 2019
Khandelwal S (2018) Thousands of MikroTik Routers Hacked to Eavesdrop On Network Traffic. https://thehackernews.com/2018/09/mikrotik-router-hacking.html. Accessed 1 May 2019

https://t.me/learningnets

Kim SY, Lee S, Yun I, Xu W, Lee B, Yun Y, Kim T (2017) Cab-fuzz: Practical concolic testing techniques for cots operating systems. In: USENIX Annual Technical Conference (USENIX ATC). USENIX, Santa Clara
Largent W, New VPNFilter malware targets at least 500K networking devices worldwide (2018). https://blog.talosintelligence.com/2018/05/VPNFilter.html. Accessed 1 May 2019
LLVM (2015) libFuzzer - a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html. Accessed 1 May 2019
Mi Smart Plug (2015). https://www.mi.com/us/mj-socket/. Accessed 1 May 2019
Muench M, Stijohann J, Kargl F, Francillon A, Balzarotti D (2018) What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In: Network and Distributed System Security Symposium (NDSS). ISOC, San Diego
NVD (2015) Common Vulnerability Scoring System (CVSS). https://nvd.nist.gov/vuln-metrics/cvss. Accessed 1 May 2019
NVD (2020) CVE-2020-15916. https://nvd.nist.gov/vuln/detail/CVE-2020-15916. Accessed 1 Dec 2020
Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H (2017) Vuzzer: Application-aware evolutionary fuzzing. In: Network and Distributed System Security Symposium (NDSS). ISOC, San Diego
rytilahti (2018) python-miio:Python library & console tool for controlling Xiaomi smart appliances. https://github.com/rytilahti/python-miio. Accessed 1 May 2019
Selenium (2004) A browser automation framework and ecosystem. https://github.com/SeleniumHQ/selenium/. Accessed 1 May 2019
Shoshitaishvili Y, Wang R, Hauser C, Kruegel C, Vigna G (2015) Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: Network and Distributed System Security Symposium (NDSS). ISOC, San Diego
Stasinopoulos A, Ntantogian C, Xenakis C (2015) Commix: Detecting and exploiting command injection flaws. In: BlackHat EU, Amsterdam
Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G (2016) Driller: Augmenting fuzzing through selective symbolic execution. In: Network and Distributed System Security Symposium (NDSS). ISOC, San Diego, California
strace (2000) strace - linux syscall tracer. https://strace.io/. Accessed 1 May 2019
Wang Z, Zhang Y, Liu Q (2013) Rpfuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing. KSII Trans Internet Inf Syst (TIIS) 7(8):1989–2009
You W, Wang X, Ma S, Huang J, Zhang X, Wang X, Liang B (2019) Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In: IEEE Symposium on Security and Privacy (S&P). IEEE, San Francisco
Zaddach J, Bruno L, Francillon A, Balzarotti D, et al (2014) Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: Network and Distributed System Security Symposium (NDSS). ISOC, San Diego, California
Zalewski M (2014) American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Accessed 1 May 2019
Zerodium (2015). https://zerodium.com/program.html. Accessed 1 May 2019
Zhang Y, Huo W, Jian K, Shi J, Lu H, Liu L, Wang C, Sun D, Zhang C, Liu B (2019) SRFuzzer: an automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities. In: Proceedings of the 35th Annual Computer Security Applications Conference. IEEE, San Juan. pp 544–556

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

Submit your next manuscript at ▶ springeropen.com

https://t.me/learningnets