RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing

Zhiqiang Wang[1], Yuqing Zhang[1,2] and Qixu Liu[2]
[1] State Key Laboratory of Integrated Services Networks, Xidian University
Shaanxi Xi'an - P.R. China
[e-mail: wangzq@nipc.org.cn]
[2] National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences
Beijing - P.R. China
[e-mail: zhangyq@gucas.ac.cn]
*Corresponding author: Yuqing Zhang

Received December 27, 2012; revised February 20, 2013; accepted June 6, 2013; published August 30, 2013

Abstract

How to discover router vulnerabilities effectively and automatically is a critical problem to ensure network and information security. Previous research on router security is mostly about the technology of exploiting known flaws of routers. Fuzzing is a famous automated vulnerability finding technology; however, traditional Fuzzing tools are designed for testing network applications or other software. These tools are not or partly not suitable for testing routers. This paper designs a framework of discovering router protocol vulnerabilities, and proposes a mathematical model Two-stage Fuzzing Test Cases Generator(TFTCG) that improves previous methods to generate test cases. We have developed a tool called RPFuzzer based on TFTCG. RPFuzzer monitors routers by sending normal packets, keeping watch on CPU utilization and checking system logs, which can detect DoS, router reboot and so on. RPFuzzer' debugger based on modified Dynamips, which can record register values when an exception occurs. Finally, we experiment on the SNMP protocol, find 8 vulnerabilities, of which there are five unreleased vulnerabilities. The experiment has proved the effectiveness of RPFuzzer.

Keywords: router security, fuzzing, TFTCG, protocol vulnerability discovering

The part of this work was presented in 7th International Conference on Communications and Networking in China. There are more than 30 percent substantial new contributions added in the manuscript. This research was supported by by the National Natural Science Foundations of China (No.60970140) and (No.61161140454) and China Postdoctoral Science Foundation(2011M500416,2012T50152). The authors would like to thank the editor and all the anonymous reviewers for their useful suggestion.

http://dx.doi.org/10.3837/tiis.2013.08.014

1. Introduction

Router is one of key devices to connect network in the Internet world, whose security plays a crucial role. The research on finding router bugs has been a hot area for several years. Since Felix Linder, a member of the hacker organization Phenoelit, attacked Cisco routers with routing & tunneling protocol in 2001 [1], research and attacks on router security have become one kind of new tendency. In 2005, Michael Lynn, a security researcher, presented a vulnerability concerned handling of IPv6 packets at the Black Hat conference [2], informally known as "Cisco gate". With his findings, attackers are allowed to execute arbitrary code remotely. Hereafter, the security of routers is increasingly focused on. At the 2008 DEFCON Conference, security expert Alex Pilosov and Tony Kapela demonstrated an attack on BGP [3], the core Internet routing protocol, which created a big stir among the industry and academia. Moreover, some vendors and individuals developed Cisco IOS debugging tools, for example, GNU debugger of IRM PLC from England [4] and modified Dynamips of Groundworks Technologies [5], with which it is more favorable for router attacks.

According to U.S. National Vulnerability Database (NVD) [6] statistics, the prevalence of router vulnerabilities is growing up shown in **Fig. 1**, and the proportion of vulnerabilities' severity is shown in **Fig. 2**. Take Cisco routers as an example, there are 1056 vulnerabilities on Cisco routers as of December 31st, 2011, of which protocol vulnerabilities account for about 72%. Most of these vulnerabilities' severities are medium or high. Thus, it can be seen that the issue of router security is becoming more and more serious and has become an important factor that affects Internet security. It's imperative to do a lot of research on the technology about discovering the vulnerabilities of routers in order to ensure network and information security.

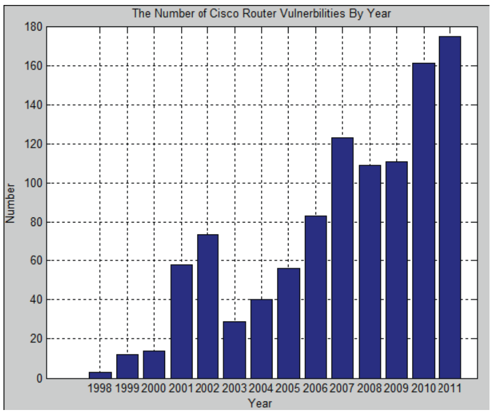


Fig. 1. The number of Cisco vulnerabilities

Although many security researchers and hackers have made remarkable progress in the study of router security technology, there are still some problems to be solved as follows.

- Previous research on router security is mostly about the technology of exploiting known flaws of routers or debugging routers. How to effectively and automatically discover router vulnerabilities becomes an urgent problem to be solved, especially protocol vulnerabilities. So far, none of existing frameworks is developed specially for testing router protocols.

- Fuzzing is an effective automatic technique to find vulnerabilities. However, current Fuzzing tools on network protocols are not or partly not suitable to testing router protocols. These tools are developed to test network applications, whose methods of monitoring and debugging targets are different from routers. In addition, single-Fuzzing that mutates a single data sample or input variable is used in these tools, whose code coverage is low and cannot find vulnerabilities resulted from the combination of multiple input variables or samples.
-

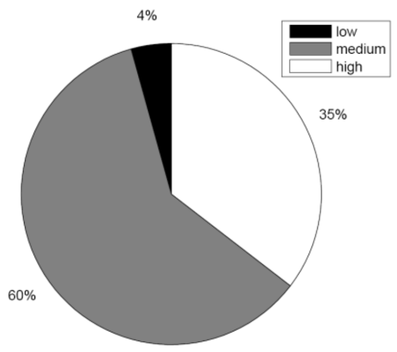


Fig. 2. The proportion of Cisco router vulnerabilities' severity

To solve the above problems, we have made a study on how to effectively find router vulnerabilities. There are two points that should be stressed here. First, our research focuses on the security of router protocols for the reason that protocol vulnerabilities account for the greatest proportion of router vulnerabilities. Second, router protocols refer to not routing protocols but all protocols supported by routers. The paper makes the following contributions:

- We design a general testing framework of router protocols to discover router vulnerabilities. It is the first integral router protocol vulnerability discovering framework. The framework is able to effectively test routers or other network devices.
- We propose a mathematical model Two-stage Fuzzing Test Cases Generator (TFTCG) to generate test cases. TFTCG consists of two stages. In the first stage, generation-based Fuzzing is combined with manual analysis and testing that analyze protocol weak points, with which we can generate effectively test cases. In the second stage, mutation-based multi-Fuzzing that mutates multiple data samples is used to generate test cases, of which samples are got from historical vulnerability data and the abnormal test cases from the first stage.
- We develop a tool called RPFuzzer, which is superior to previous network protocol testing tools on the strategy of test case generation and the methods of monitoring and debugging routers. RPFuzzer is developed based on the above framework and the model TFTCG. The monitor of RPFuzzer uses three methods to monitor routers, including sending normal test cases, keeping watch on CPU utilization of routers and checking system logs, which can detect DoS vulnerabilities, router reboot, zombie process and so on. RPFuzzer's debugger is developed modified Dynamips [5] which can record register values when an exception occurs that is helpful for researchers to prepare related solutions to fix flaws.

The remainder of the paper is organized as follows. In section 2, we give a review of the related work. Section 3 introduce a general way of discovering vulnerabilities on network protocols and present our method in views of network protocols applied in router. Section 4 describes the design and implementation of the router protocol vulnerability discovering framework based on multi-Fuzzing. In section 5, we do some experiments on the SNMP protocol. Experimental evaluations are discussed in section 6. Finally, conclusions and future work are given.

2. Related Work

In the past decade, vulnerability discovering methods and related study on routers have been advancing rapidly, and most research focuses on Cisco router. Felix Linder from Phenoelit analyzes several IOS vulnerabilities and various exploitation techniques [7]. Michael Lynn presented a technique to take control of an IOS-based router, which is achieved by means of a buffer overflow or a heap overflow, two types of memory vulnerabilities [2]. Gyan Chawdhary and Varun Uppal proposed a method to debug Cisco IOS and write shellcodes with GNU debugger, which makes it easier to attack routers [4]. Felix Linder put forward an exploit technique that uses fragments of code from the ROMMON for reliably exploiting buffer overflows in Cisco routers [8][9], which solves a key problem of setting the return address of shellcodes. A twostage attack strategy against Cisco IOS was presented by Ang Cui et al, which can make two unique multi-stage shellcodes capable of reliable execution within a large collection of IOS images on different hardware platforms [10]. Sebastian Muniz and Alfredo Ortega presented a tool which facilitates debugging and reverse engineering process of Cisco IOS by allowing the integration with most used existing debugging and disassembler tools such as GDB and IDA Pro [11].

References [2] and [7] only lay stress on how to exploit two types of vulnerabilities. References [4] and [11] put stress on how to debug routers. References [8] and [9] introduce the method of writing shellcodes. Reference [10] makes Cisco IOS diversity not difficult to reliably execute shellcodes. The above-mentioned references just emphasize how to take advantage of known vulnerabilities, debug routers and write reliable shellcodes, none of which puts forward a general framework how to discover router vulnerabilities effectively. Fuzzing is a kind of software vulnerability mining technique and is able to find network protocols bugs effectively, on which the research is relatively mature. Miller et al. [12] first introduced fuzz testing that inserts fault data randomly into the input of UNIX system utilities using data mutation. Reference [13] introduce the definition of Fuzzing, the methodology, intelligent &unintelligent fuzzers, common Fuzzing problem(various types of validation) and application behaviors. There are two forms of fuzzing program, namely generation-based and mutation-based [12][13]. Mutation-based Fuzzing constructs test cases by mutating the fields of a given and normal sample in advance. The efficiency of this method is low on account of not considering the constraint relations between various input variables or vulnerable points. Generation-based Fuzzing constructs test cases according to a specification which describes the file format or network protocol [14]. Test cases constructed by this method are more valid than that constructed by mutationbased Fuzzing, because the test cases are constructed on the basis of the specification. However, automated testing for generation-based Fuzzing, which need manual analysis to get the knowledge of of tested protocols or applications, is not as easy as that for mutation-based Fuzzing. References [15] and [16] propose multipledimension mutation and generation(m&g), which means mutating multiple input element or vulnerable points at a time to form a test case. Multiple-dimension mutation and generation can effectively find bugs caused by multiple vulnerable points.

At present, there are a lot of famous Fuzzing frameworks presently, such as SPIKE [17], Peach [18], Sulley [19], Autodafe [20] and GPF [21], of which SPIKE, Peach and Sulley belong to semi-automatic tools, while Autodafe and GPF belong to automated tools. The technology of automatic analysis on network protocols is still immature at present, namely the semi-validity of generated test cases is still not high. So Autodafe and GPF are not as efficient as automatic tools. So we just consider semi-automatic tools. SPIKE is a well-known Fuzzing tool, which adopts generation-based strategy. It allows you to quickly create network protocol stress testers. However, the number of test cases generated by SPIKE is small, and there is no a monitor. Peach is a cross-platform Fuzzing framework, whose data generation strategy is based on mutation with the analysis on tested protocols and known vulnerabilities called knowledgebased Fuzzing technology. Similarly, Peach is deficient in effective monitoring routers. Sulley is a fuzz testing framework consisting of multiple extensible components [22]. Different from previous fuzzers that solely focus on data generation, Sulley has not only impressive data generation but also instruments and monitors the health of the target, capable of reverting to a known good state using multiple methods, which improves automatic degree. Nevertheless, Sulley's monitor is partly applicable to routers and lacks a debugger, the method of whose monitor adopts is monitoring the session between processes. Sulley'monitor may missed exceptions that CPU utilization is less than 100%. Moreover, above tools only consider Fuzz testing on a single data sample set or input variable [16], which leads to low code coverage and the effect of mining vulnerabilities is not stable.

So far, there is not a well-rounded framework designed to discover vulnerabilities about router protocols. Besides, generating test cases and monitoring tested targets are also needs to be improved.

3. Methodology

In this section, the principle and process of network protocol testing based on Fuzzing is introduced. Afterwards, our method is described in detail.

3.1 Fuzzing Test on Network Protocols

Fuzzing is a well-known black-box technique for the security testing of applications [22]. The objective of Fuzzing test on network protocols is to test whether all kinds of network devices or related applications have security vulnerabilities or not. The principle is to send malformed testing data to targets through Socket APIs and monitor the exceptions appeared in targets [14][16][22][23][24]. The testing procedure can be divided into five steps [22]. Firstly, identify the target to be tested and get more details about the target. Secondly, identify inputs and potential variables, such as headers, filenames, environment variables, etc. Then configure targets preparing for testing. After generating Fuzzing data, we can execute Fuzzing data and monitor for exceptions. Lastly, once finding a fault, it is necessary to determine whether the bug discovered can be exploited.

3.2 Fuzzing Test on Router Protocols

Before taking a glimpse of our method, we firstly introduce single-Fuzzing and multi-Fuzzing. Previous methods in section 2, applied in some Fuzzing tools such as Peach, Sulley, can only be called single-Fuzzing that just considers a single sample data or mutates one field at a time to generate a test case. Certain vulnerabilities can only be triggered by some special combination of multi-dimensional input, so they will be missed by single-Fuzzing [16], moreover, whose efficiency is not stable. Multi-Fuzzing that mutates multiple input variables is first proposed in Reference [16], which adopts mutation-based multi-Fuzzing and genetic algorithm to mining soft vulnerabilities. However, the method in Reference [16] needs to establish the relationships between input elements and insecure functions by static analysis on the source code. It is inapplicable to test routers. In addition, this approach can take an inordinately long time to generate valid data for protocols that contain TLV style fields [22]. In our strategy, we just adopt multi-Fuzzing to mine vulnerabilities and make the efficiency of discovering flaws stable, in other words, we find *m* bugs in a test and may find *n* bugs in another test($m \neq n$).

The method that the paper proposes can be divided into two stages. The first stage adopts a combination of manual analysis and testing and Fuzzing based on generation. The second stage adopts multi-Fuzzing based on mutation with the sample data from the first stage and historical vulnerability data. The flowchart of the method is shown in **Fig. 3**.

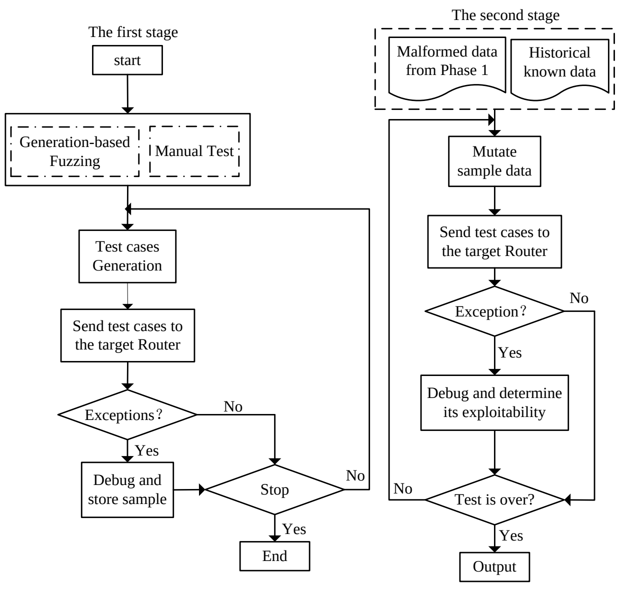


Fig. 3. The Flowchart of our method

The First Stage: With manual analysis and testing on router protocol and the cause of related historical vulnerabilities where bugs may exist, we can gain lots of knowledge about vulnerable points. Then, we generate test cases with generation-based Fuzzing based on obtained knowledge. The generation of test cases in the first stage is shown in **Fig. 4 (1)**. Suppose a packet consists of 5 fields, and the vulnerable fields are Fields 2 and 4. We construct test cases by replacing the vulnerable fields with malformed data and other fields with normal data. When exceptions occur during testing routers, a debugger is invoked to debug breakpoints and store malformed data that will be used in the second stage. Afterwards, recover to be normal for continuing testing. The first stage use the incorporation of the two methods we called knowledgebased Fuzzing, which can improve the semi-validity of test cases and efficiency of testing.

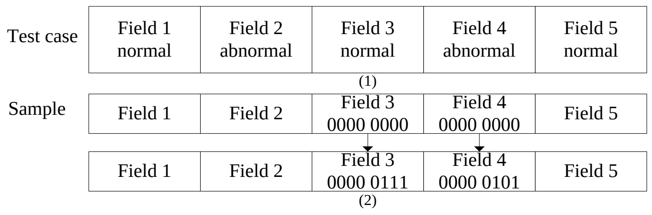


Fig. 4. Generation-based and mutation-based multi-Fuzzing

The Second Stage: Mutation-based multi-Fuzzing is used and the construction of test cases is shown in **Fig. 4(2)**. Before the mutation operation, a sample should be provided. Suppose a sample consists of 5 fields, Fields 3(0000 0000) and 4(0000 0000) are the fields to be mutated. Field 3 may be mutated into (0000 0111) and Field 4 may be mutated into (0000 0101). Mutation-based multi-Fuzzing means that we mutate two or more vulnerable points of a sample at a time. The sample data is obtained from the first stage and historical data which leads to abnormality of routers, the reason for which is that previous malformed values is likely to trigger an old or new bug and multiple-dimension Fuzzing needs a variety of samples to ensure stable efficiency. For example, suppose a string ”AAA...” can trigger a buffer overflow vulnerability shown in **Fig. 5**, the string ”BBB...” might as well cause this bug. The historical data is selected from NVD [6],CVE [25] and so on. The approach to select the sample data not only enhances the code coverage [15] and but also improves the vulnerabilities finding efficiency. Upon finding abnormal information, the debugger will be called to debug the breakpoints and record debugging information. Then continue testing until the test is over.

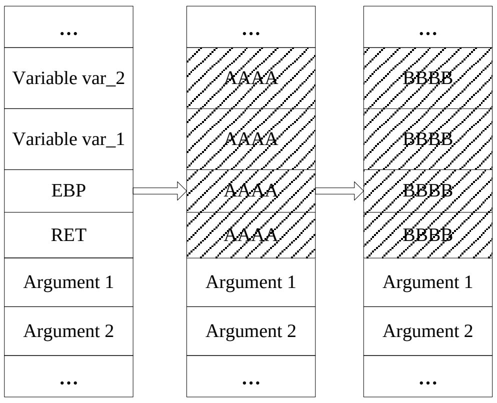


Fig. 5. An example: a buffer overflow

3.3 TFTCG

To understand two-stage strategy of generating test cases above, we introduce a universal mathematical system TFTCG, namely Two-stage Fuzzing Test Cases Generator. TFTCG is as follows:

TFTCG = (**F**, **MDB**, **G**, **SDB**, **M**, **OP**, **Result**)

OP = {**fuzz_generator, single_mutator, multi_mutator, CalChsum**}

Result = {**Testcases**}

fun1: (**F**,**C**) × **MDB→G**

fun2: **SDB** × **F→M**

F, a vulnerable field set, **F** = $\{f_1, f_2, ..., f_n\}$, f_i denotes a weak field in network protocols, $1 \leqslant i \leqslant n$, such as version, PDU type or source addresses, n is the number of fields.

C, an attribute set of relative fields, **C** = $\{c_1, c_2, ..., c_n\}$, c_i denotes an attribute of c_i field, $1 \leqslant i \leqslant n$, such as field types, range of values.

MDB, a database of malformed data fragments, **MDB** = $\{M_1, M_2, ..., M_r\}$, M_i denotes a type of malformed data in database **MDB**, $1 \leqslant i \leqslant r$, such as format string data.

G, a flag set of generation operations, G = $(g_{ij})_{n \times r}$, $g_{ij} = 0$ or 1, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant r$. When $g_{ij} = 1$, it means that we will generate test cases with malformed data M_j at the field f_i, 0 not.

SDB, a set of sample data, SDB = $\{S_\alpha, S_\beta\} = \{ s_1, s_2, ..., s_q \}$, q is the number of samples. S_α denotes sample data from the first stage, and S_β denotes sample data gathered from NVD, CVE and so on.

M, a flag set of mutation operations, **M** = $(m_{ij})_{n \times q}$, $m_{ij} = 0$ or 1, $m_{ij} = 0$ or 1, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant q$. When $m_{ij} = 1$, it means that we will mutate f_i field of S_j randomly, 0 not.

OP, denotes a relative operations set. **single_mutator** and **multi_mutator** represent single mutaion and multiple mutation respectively. **fuzz generator** represents generating test cases with generation-based Fuzzing. The function **CalChsum()** is designed to compute checksum if needed (flag=1).

Testcases, denotes test cases generated in the above two stages. **Testcases** = $\{T_1, T_2\}$, T_1 denotes test cases generated in the first stage, and T_2 in the second stage.

Function ***fun1*** denotes mapping $((f_i, c_i), M_j)$ to 0 or 1 according to c_i, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant r$. We can get generation matrix **G**, i.e. the strategy of generating test cases in the first stage.

Function ***fun2*** denotes mapping (s_i, f_j) to 0 or 1, $1 \leqslant i \leqslant q$, $1 \leqslant j \leqslant n$. We can get mutation matrix **M**, i.e. the strategy of mutating test cases in the second stage.

The algorithm of generating test cases is described in **Table 1**. First, we need to initialize matrixs **G** and **M**, each element of which is assigned 0 or 1. For each g_{ij} in **G**, we construct test cases taking advantage of generation-based Fuzzing when g_{ij}=1. After the first stage, we can get S_α from the first stage. In the second stage, for each m_{ij} in **M**, we construct test cases taking advantage of mutation-based single-Fuzzing and multiple-Fuzzing, when m_{ij}=1.

Table 1. The algorithm of TFTCG

```
Algorithm 1
Input: F, MDB, SDB
Output: Testcases
1  Begin
2  //Initialization
3  calculate matrix G and M;
4  Testcases = Φ;
5  for each $g_{ij}$ in G do
6     Testcases = Testcases ∪ fuzz_generator($f_i$, $M_j$, $g_{ij}$, flag);
7  //We can get $S_\alpha$ from the first stage.
8  end
9  for each $m_{ij}$ in M do
10    for each $s_j$ in SDB do
11       if $m_{ij}$ then
12          Testcases = Testcases ∪ CalChsum(single_mutator($f_i$,$s_j$), flag);
13       end
14       Testcases = Testcases ∪ multi_mutator( );
15     end
16  end
```

In **Table 1**, the functions fuzz_generator( ) and multi_mutator( ), which denote generating and mutating test cases, will be described in **Table 2** and **Table 3**.

Algorithm 2: Suppose there are n vulnerable points existing in tested protocol. Let m denotes the dimension of mutation-based multi-Fuzzing, and **M** = |**MDB**| denotes the number of values in the database **MDB**. In the first stage, the function *replace(T, f_i, b_j)* denotes replacing with data b_j at the place a_i of the packet T, b_j from the database of malformed data **MDB**, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant M$. In other words, the function *replace( )* means constructing test cases with malformed data in the database **MDB**. For example, in **Fig. 4 (1)**, we construct a test case by replacing field 1, 3 and 5 with "normal" data and replacing fields 2 and 4 with "abnormal" data. The function *CalChsum()* is designed to compute checksum if needed (flag=1). For example, to test the protocol ARP, we should compute the checksum when constructing an ARP packet. If we construct a SNMP packet and send it by a socket API, there is no need to compute checksums.

Table 2. The algorithm of fuzz_generator

```
Algorithm 2
Input: F, MDB, g_ij
Output: T_1(test case set in the first stage)
1   Begin
2     T_1 = Φ;
3     T = T_default;// T_default denotes the default sample.
4     for(i = 1; i ⩽ n; i++) do
5         if g_ij=1 then
6             for each M_k in M_1, M_2, …, M_r do
7                 for each b_j in M_k do
8                     T_1 = T_1 ∪ replace(T, f_i, b_j);
9                 end
10                for each test case t in T_1 do
11                    CalChsum(t, flag)
12                //If flag = 1, calculate the checksum of t, 0 not.
13                end
14            end
15        end
16    end
17    return T_1;
18  end
```

Algorithm 3: In the second stage, we first select samples from the sample database **SDB**. The function *select_sample(count, **SDB**)* means selecting *count* samples from **SDB**. Then we construct new test cases by mutating the samples. The function *mutate($T_{Samples}$, c, d, N)* denotes mutating $T_{Samples}$ N times at the fields c and d of the sample $T_{Samples}$, $1 \leqslant c, d \leqslant n$, $T_{Samples}$ denotes sample data. For example, in **Fig. 4(2)**, we construct a test case by mutating fields 3 and 4 of a sample and keeping other fields of the sample unchanged. Afterwards, we compute the checksums of test cases as the first stage do. By the way, m=2 in algorithm 3. When a higher value of m will bring about the input combination explosion that a mass of test cases is created in the generation process. We don't discuss the issue on the optimization of the dimension m.

Table 3. The algorithm of multi_mutator

```
Algorithm 3
Input: F, SDB
Output: T_2_multi
1   Begin
```

```
$T_2$_multi $= \Phi$;
for ($count = 0$; $count < |\mathbf{SDB}|$, $count$ ++) do
    $temp = 1$;
    $T_{samples} = select\_sample(count, \mathbf{SDB})$;
end
for ($c = 1$; $c \leqslant n$; $c$++ ) do
    while ($temp \leqslant N$) do
        $T_2$_multi = $T_2$_multi $\cup$ CalChsum( $mutate(T_{samples}, c, d), flag$)
    end
end
end
```

4. Designs AND Implementation

According to the methodology in section 3, we design a vulnerability discovering framework in view of router protocols, and develop a tool called RPFuzzer based on the architecture and TFTCG with the Python language. The architecture of RPFuzzer consists of seven parts: script parser module, test cases generator module, tester module, monitor module, debugger module, verifier and output module and data import module. The architecture is illustrated in **Fig. 6**. We will introduce and analyze the first six modules.

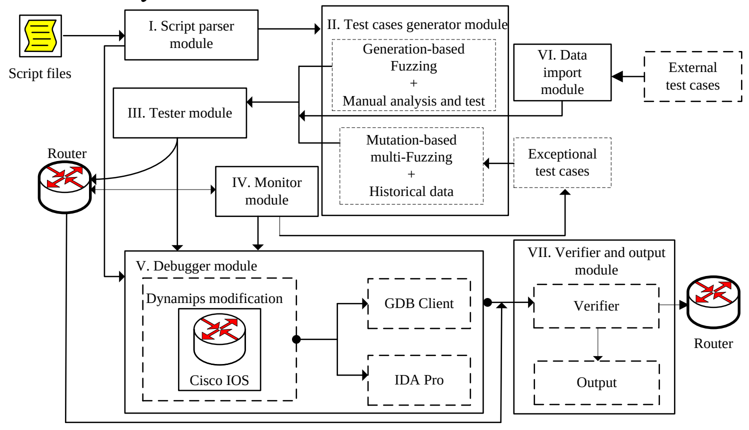


Fig. 6. The architecture of RPFuzzer

4.1 Script Parser Module

Script files that RPFuzzer parses are comprised of two types of files, protocol script files and configuration files. Protocol script files are used to describe protocol specifications, including protocol types, ports, fields, vulnerable points etc. Manual analysis on the tested protocol is indispensable to obtain above information. Configuration files contain the commands that create and configure a virtual network adapter, the path of Cisco IOS images, GDB debugging Port, the storage location of log files, the path and configuration information of Dynamips.

Script Parser Module can parse above scripts files, and obtain protocols format and related configuration information. The configuration information covers router protocols specification and debugging requirements. The former is used to configure the target protocol before a test can be performed. The latter is ready for debugging router. When exceptions occur, RPFuzzer will start the debugger with the debugging configuration information, and preserve current register values at the breakpoint.

4.2 Test Cases Generator Module

Test Cases Generator Module just as its name implies is to generate test cases, which can be divided into two stages shown in **Fig. 3**. Based on the mathmetical model TFTCG, we can generate test cases with malformed data at the points, which may exist bugs.

Different protocol has different vulnerable points. Taking the TFTP protocol as an example, vulnerable points may include long filename and directory traversal [26], which don't exist in the SNMP protocol. Therefore, it is necessary and important to set the fields to be generated or mutated, which is determined by manual analysis. To generate test cases, we design a database **MDB** of malformed data, with which we can generate or mutate the weak fields. The database consists of boundary value, overlong character strings, separators, format strings and so on. The above sensitive data is illustrated in **Table 4**. In addition, when analyzing certain protocol, another sensitive data about tested protocols can also be added into the database **MDB** for testing the same protocol. According to the database **MDB** of malformed data and the vulnerable points from manual analysis, we can generate test cases automatically based on TFTCG.

Table 4. The Database of Malformed Data

Type	Malformed data
integer	0x00, 0x0000, …; 0xFF, 0xFFFF, 0xFFFFFFFF, … ; 1, 2, 3, … ; 0x7F, 0x7FFF, 0x7FFFFFFF, … ; 0x80, 0x81; 0x8001, 0x80000001, …; 0xFF-1, 0xFFFF-1, 0xFFFFFFFF-1, 0xFFFFFFFF-2, 0xFFFFFFFF-3, …; 0xFFFFFFFF/2, 0xFFFFFFFF/2-1, 0xFFFFFFFF/2-2 ,…
character string	Overlong strings: AAAAAA… ; BBBBBB… ;
nonalphanumeic characters	field delimiters including tabs and spaces; others: !, @, #, $, %,ˆ , &, *, (,), -, , =, +, {, }, \, ;, :, |, ”, ’, <, >, /, ?, and so on;
format string	%d, %x, %s, %n and derived strings, such as %n%s%n%s, …, %s%s%s%s … , etc.
character conversion	0x0, 0xFE, 0xFF, 0xef, 0xbb, 0xbf, 0xfe, 0xff, 0x10FFFF, overlong “%2f” and “%5c”
directory traversal	~/, /··, ··/··/, \·, \··, ··\··\, and derived strings

4.3 Tester Module

The target routers that RPFuzzer tests are divided into two types, virtual routers and real router devices. The flow chart of the tester module is shown in **Fig. 7**. Firstly, select the target, virtual or physical device. Then configure the target according to protocol script files and configuration files. If the target is a virtual router simulated by Dynamips, launch Cisco IOS and configure the tested protocol. When an exception occurs after sending test cases to the target router, invoke the debugger to record the values of registers and save the malformed test cases. Then recover to normal for continuing testing till the end. If the target is a physical router or a router simulated by other emulators, record the malformed test cases while an exception occurs after booting up and configuration.

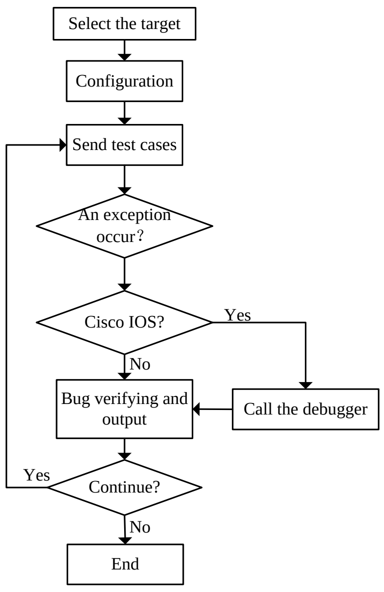


Fig. 7. Testing procedure

4.4 Monitor Module

Generally, there are three methods to monitor routers: monitoring CPU utilization, sending monitoring data and checking system log.

(1) CPU Utilization.

By Monitoring CPU utilization, we can effectively detect DoS attack caused by the CPU utilization abnormalities of the process that handles tested protocol data. But this method has many limitations. For example, it is not applicable to cases that routers crash or reboot.

(2) Sending Monitoring Data.

We can also detect the abnormalities of routers by sending monitoring data. For instance, we can either send normal SNMP messages to routers at regular interval or execute PING command, i.e. send ICMP packets to monitor routers. This approach can detect router crash or reboot and has high automation and credibility. However, the exception caused by abnormal CPU utilization, we call it “mild-DoS attack”, could not be detected by this method.

(3) System Log.

System log can record the activities in router system, including initialization, reboot, configuration and some error information. By checking system log, we can detect router crash, reboot, zombie process etc [23]. But this method cannot detect “mild-DoS attack”and traffic anomaly, furthermore, the method doesn’t work in real time.

In order to monitor routers better, the combination of three methods is adopted in RPFuzzer, which can detect router crash, reboot, “mild-DoS attack”and so on.

4.5 Debugger Module

The debugger is developed on the basis of modified Dynamips [5], which is a tool that facilitates debugging and reverse engineering process of Cisco IOS by GDB [27] or IDA Pro [28]. GDB is only used in RPFuzzer’s debugger, while IDA Pro in dashed frame is left for expansion later. The debugging procedure is illustrated in **Fig. 8**. When an exception occurs during testing, the debugger firstly record the number of malformed test case and execute “x/i $pc”and “info register”to log “breakpoint”, namely the values of all registers, which may be used in vulnerability exploits. Then reboot and configure the router, continue testing beginning with next test case till the end.

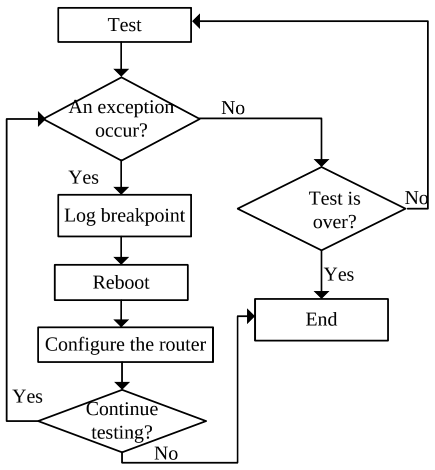


Fig. 8. The procedure of debugging routers

4.6 Verifier and Output Module

There are two kinds of verifiers, virtual and real.

(1) Virtual

Virtual verifier mainly analyzes breakpoints and anomaly including router crash, reboot, "mild-DoS attack"and zombie process on the virtual router emulated by Dynamips. We determine whether the exception exists or not by resending the recorded test cases to virtual router and checking up the registers. If the registers are abnormal, system log describes abnormal records or CPU utilization is abnormal, there are flaws existing in routers.

(2) Real

Real verifier is aimed at verifying bugs in physical router. It sends the malformed test cases that bring about exceptions to corresponding physical routers. Due to the financial constraints and other factors, it is infeasible for us to test all physical devices. During the experiments, we just verify the flaws on several physical routers. Nevertheless, the exceptions and related records can be submitted to router vendors, who will further verify the exceptions. Real verifier could make up the drawback of virtual verifier that the emulation of routers is not 100% exact [11].

After confirming the vulnerabilities, we explicitly output vulnerability information in detail.

5. Experiments on SNMP

To validate the effectiveness of RPFuzzer, we do experiments on SNMP. The following will describe the procedure of testing SNMP in particular.

5.1 SNMP Protocol

Simple Network Management Protocol (SNMP) is an Internet-standard protocol for managing devices on IP networks. It is used mostly in network management systems to monitor network-attached devices for conditions that warrant administrative attention [30]. There are four versions of SNMP, and different versions of SNMP have somewhat different messages. There are several core PDUs of SNMP: GetRequest, GetNextRequest, SetRequest, GetBulkRequest, Response and Trap. Five messages of SNMP are shown in **Fig. 9**, and

Message formats are illustrated in **Fig. 10** [29]. The messages formats of SNMPv2c, SNMPv3 [30][31] and other versions are not introduced particularly.

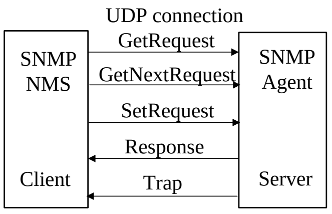


Fig. 9. Five messages of SNMPv1

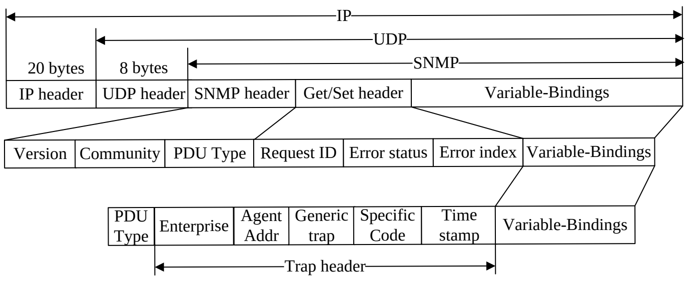


Fig. 10. The message formats of SNMPv1

5.2 SNMP Vulnerable Point Analysis

The vulnerable points will be analyzed manually and empirically in this section. Fuzzing has particular limitations in types of vulnerabilities it will find, such as access control flaws, poor design logic, backdoors, memory corruption and multistage vulnerabilities [22]. For this reason, we just consider the vulnerabilities that Fuzzing can find. In accordance with the analysis on historical vulnerabilities and the SNMP protocol, there are five types of vulnerable points on SNMP.

(1) ASN.1 BER parse

The SNMP protocol specifies ASN.1 with BER as its required encoding scheme. Each data element is encoded as a type identifier, a length description, the actual data elements and where necessary, an end-of-content marker.

About BER rules, possible vulnerable points are invalid encodings, including invalid types, abnormal lengths and malformed values [32]. An invalid type/length/value encoding means replacing right encodings with malformed encodings. For example, the encoding of Integer is 0x02, we can replace it with 0x04(OCET String) or 0x05(NULL).

2) Integer overflow

Integer overflow is caused by malformed Integer values including boundary value, large Integer number and other values than Integer, besides, the transformation from signed Integer value to unsigned may leads to anomaly. E.g. large Integer number $2^{256}+1$ or $(-2^{256})-1$ is likely to cause Integer overflow. Although Integer overflow never happened on SNMP before, it is indispensable for Fuzzing test.

(3) Buffer overflow

Buffer overflow is caused by incorrect input strings, which include long character strings and format character strings. For SNMP, vulnerable points about buffer overflow could be the field Variable-Bindings according to historical statistics. So we can test this field with zero-length Object Identifier (OID), overlong single or multiple format character strings, overlong OIDs with many branches.

(4) Empty packets

Empty packets include empty UDP packets, empty IP packets, and empty SNMP packets. The data of above packets is tested with 0x00. There are a lot of vulnerabilities caused by empty packets. Such as CVE-2001-0566, its root cause is to send an empty packet to port 161(SNMP).

(5) A large number of packets

Sending a large number of packets to routers could allow attackers to cause a denial of service or gain privileges, such as CVE-2002-0012 and CVE-2002-0013. This is a significant cause for denial of service attack. Hence, we should test routers with a large number of all kinds of SNMP packets.

5.3 Test Cases Generation

According to the analysis in section 5.2, we can determine the fields and malformed data to be tested described as follows.

BER test: Each data element is encoded with TLV encodings. Hence, we could test all the fields in SNMP packets. The test covers three parts: type, length and value. Type and length can be generated with the data in **Table 5**, while value will be introduced particularly.

Table 5. The data for BER test

Name	Malformed data (hex) (R denotes a random data)
type	0x02, 0x04, 0x06, 0x05, 0x30, 0x40, 0x41, 0x42, 0x43, 0x44, 0xRR
length	0x0, 0xFF, 0xFFFF, 0xRRRR

Integer overflow: This part is aimed at the fields whose type is Integer. The fields cover SNMP header, Get/Set/GetNext/GetBulk header, Trap header and some fields in SNMPv3, shown in **Table 6**. These fields can be generated with the data from **MDB**.

Table 6. The fields for Integer overflow test

Name	Fields
SNMP header	Version, PDU Type
get/set/getnext request and getBulk header	Request ID, Error-status, Error-index
Trap header	Generic-trap, Specific Code, Time stamp
Other fields (SNMPv3)	globalData, msgFlags, msgID, msgMaxSize

Community name test: The vulnerabilities about community name are discovered more than once in the past, such as CVE-2008-1320. We can generate the community name field with the character string data in the database MDB of malformed data. Variable-Bindings field: There are lots of vulnerabilities about this field, the reasons for which are overlong OID, zero-length OID, format string OID and so on. We specially design some malformed data in this field, shown in **Table 7**.

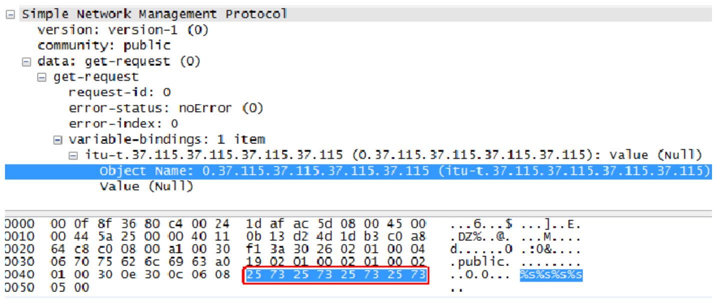
A large number of Request/Response packets: A large number of Request or Response packets could cause a massive amount of CPU utilization, which can lead to crash or reboot of routers. We can send a great number of Get/Set/GetNext/GetBulk Request, Response or Trap packets malformed or normal to routers. It is important to note the malformed data in **Table 5** and **Table 7** should be added into the database **MDB** of malformed data , with which we will test the SNMP protocol used in routers and other applications without analyzing it next time.

Table 7. The Malformed data about Variable Bindings

Name	Malformed data
Overlong single OID	$(1.3.6.1.2.7.5.1.1.181.23.34.14.23)^n$, $(1.1.1.1.1.1.1.1.1.1.1.1.1.1)^n$, *n* denotes number of OIDs
Overlong multiple OID	Two or more OID, such as $[(1.3.6.1.2.7.5.1.1.181.23.34.14.23)^n+(1.3.6.1.2.7.5.1.1.181.23.34.14.23)^m]^k$, *n*, *m* and *k* denote number of OIDs
Trap header	Generic-trap, Specific Code, Time stamp
Other fields (SNMPv3)	globalData, msgFlags, msgID, msgMaxSize

5.4 Sample Data about SNMP

In the second stage for generating test cases, historical vulnerabilities data about SNMP is required as sample data. We gather 20 vulnerabilities about SNMP in network devices of Cisco, removing the vulnerabilities that are not suited to Fuzzing test. We select one sample packet from vulnerabilities of the same kind. For example, there are 5 packets that can cause the same bug, and we select one packet as a sample. The identifiers of selected vulnerabilities are CVE-2001-1097, CVE-2002-0012, CVE-2002-0013, CVE-2003-1002 and CVE-2004-0714. More than one sample is employed in the second stage, which can keep test efficiency stable and enhance code coverage. In addition, we gain 10 abnormal samples from the first stage, which can lead to denial of service attacks including exceptions caused by empty packet, malformed OIDs. Both of the samples will be used as samples in mutation-based Fuzzing. We give a malformed test case whose OID is "%s%s%s%s" from the first stage shown in Fig. 11.



```
⊟ Simple Network Management Protocol
    version: version-1 (0)
    community: public
  ⊟ data: get-request (0)
    ⊟ get-request
        request-id: 0
        error-status: noError (0)
        error-index: 0
      ⊟ variable-bindings: 1 item
        ⊟ itu-t.37.115.37.115.37.115.37.115 (0.37.115.37.115.37.115.37.115): Value (Null)
            Object Name: 0.37.115.37.115.37.115.37.115 (itu-t.37.115.37.115.37.115.37.115)
            Value (Null)

0000  00 0f 8f 36 80 c4 00 24  1d af ac 5d 08 00 45 00   ...6...$ ...]..E.
0010  00 44 5a 25 00 00 40 11  0b 13 d2 4d 1d b3 c0 a8   .DZ%..@. ...M....
0020  64 c8 c0 08 00 a1 00 30  f1 3a 30 26 02 01 00 04   d......0 .:0&....
0030  06 70 75 62 6c 69 63 a0  19 02 01 00 02 01 00 02   .public. ........
0040  01 00 30 0e 30 0c 06 08  25 73 25 73 25 73 25 73   ..0.0... %s%s%s%s
0050  05 00                                              ..
```

Fig. 11. A malformed test case

5.5 Experimental Setup

Different tested targets have different setups. We give two types of setups illustrated in Table 8. The former are virtual routers emulated by Dynamips, the latter are physical routers or virtual routers emulated by other simulators.

Table 8. Environment Setups

Target	Cisco router	Huawei router
IOS/VRP	12.x	VRP x.x
Platform	C26xx	Nexx
System Version	CentOS-6.0	CentOS-6.0
Tested Protocol	SNMP	SNMP
Adapter	tap device	tap device
GDB Port	4321	4321

According to the setups above, we can build the following environments to test routers, illustrated in **Fig. 12** and **Fig. 13**. The former is for Cisco routers, and the latter is for Huawei or other routers. The difference between two environments is whether call the debugger when an exception occurs, the reason for which is that dynamips can only emulate Cisco routers.

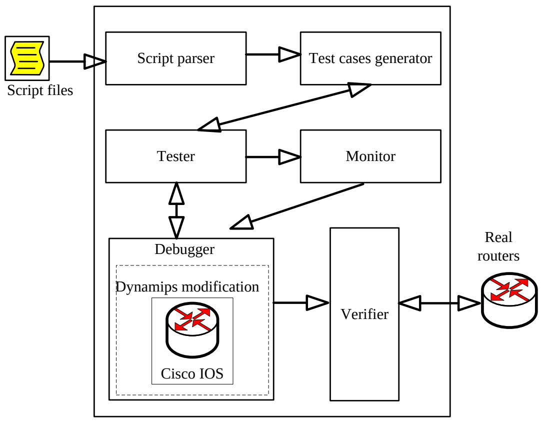


Fig. 12. Test environment of Cisco routers

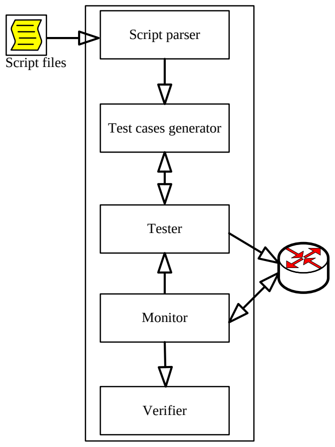


Fig. 13. Test environment of Huawei routers

6. Evaluation

In this section, we will provide test results of experiments, and evaluate the performance of RPFuzzer by comparing with other tools.

6.1 Test Results

Through testing two kinds of routers, we found 8 vulnerabilities, including 5 unreleased vulnerabilities. Testing results are shown in **Table 9**.

Table 9. Testing results

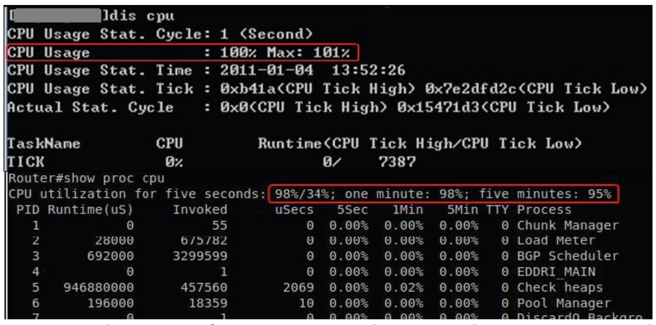
Name	Description	CVE
Cisco router 12.x	Empty UDP packet (SNMPv1)	CVE-2001-0566 CVE-2001-1097
	A large number of GetRequest, SetRequest, GetNextRequest, GetBulk Request (SNMPv1)	CVE-2002-0012 CVE-2002-0013
	Empty UDP packet (SNMPv2c/v3)	unrealsed
	A large number of GetRequest, SetRequest, GetNextRequest, GetBulk Request (SNMPv2c/v3)	unrealsed
Huawei router Nexx	Empty UDP packet (SNMPv1)	unreleased
	A large number of GetRequest, SetRequest, GetNextRequest, GetBulk Request and Trap(SNMPv1)	unreleased
	Empty UDP packet (SNMPv2c/v3)	unreleased
	A large number of GetRequest, SetRequest, GetNextRequest, GetBulk Request (SNMPv2c/v3)	unreleased

The vulnerabilities in **Table 9** can be divided into two categories: empty UDP packet and a large number of SNMP request packets.

(1) A large number of Request Packets

We create a denial of service by sending a large number of Get/Set/GetNext/GetBulk

Request, Trap/Response packets to port 161(SNMP) when SNMP is enabled, with the field variable-bindings tested with overlong or nested OIDs. We just provide abnormal CPU utilization of routers shown in **Fig. 14**, for registers of tested routers is normal. CPU utilization of a Cisco router is shown in the above and that of a Huawei router in the below.



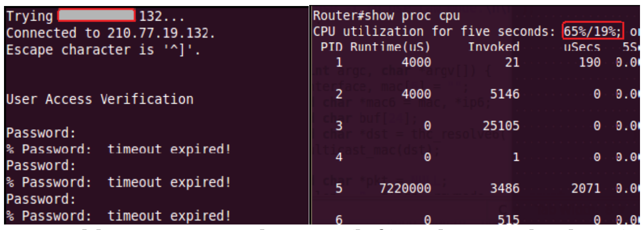
```
[[redacted]]dis cpu
CPU Usage Stat. Cycle: 1 (Second)
CPU Usage            : 100% Max: 101%
CPU Usage Stat. Time : 2011-01-04  13:52:26
CPU Usage Stat. Tick : 0xb41a(CPU Tick High) 0x7e2dfd2c(CPU Tick Low)
Actual Stat. Cycle   : 0x0(CPU Tick High) 0x15471d3(CPU Tick Low)

TaskName         CPU        Runtime(CPU Tick High/CPU Tick Low)
TICK              0%                   0/     7387
Router#show proc cpu
CPU utilization for five seconds: 98%/34%; one minute: 98%; five minutes: 95%
 PID Runtime(uS)     Invoked      uSecs    5Sec   1Min   5Min TTY Process
   1           0          55          0  0.00%  0.00%  0.00%   0 Chunk Manager
   2       28000      675782          0  0.00%  0.00%  0.00%   0 Load Meter
   3      692000     3299599          0  0.00%  0.00%  0.00%   0 BGP Scheduler
   4           0           1          0  0.00%  0.00%  0.00%   0 EDDRI_MAIN
   5   946880000      457560       2069  0.00%  0.02%  0.00%   0 Check heaps
   6      196000       18359         10  0.00%  0.00%  0.00%   0 Pool Manager
   7           0           1          0  0.00%  0.00%  0.00%   0 Discard0 Backgro
```

Fig. 14. CPU utilization of Cisco router(above) and Huawei router(below).

(2) Empty UDP packet

A flood of empty UDP packets can result in a "Mild-DoS" to tested routers whose CPU utilization is less than 100%, such as 65% in **Fig. 15** (left). For the purpose of illustrating this type of vulnerabilities, we try to telnet the tested router. Timeout expired when input a password is needed shown in **Fig. 15** (right).

Furthermore, we test Response Time (RT) and Packet Loss Rate (PLR) of the tested router under a "Mild-DoS"shown in **Fig. 16**. Response Time is uneven and some parts are unconnected for the reason that request timed out or ICMP packet is lost. The blue border marks that request timed out, while PLR is about 9% marked by a red border.



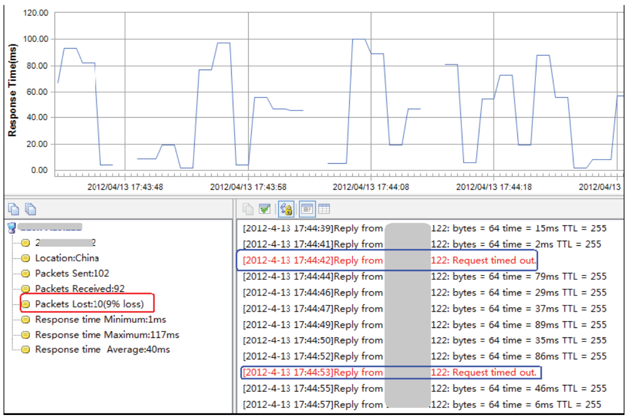
```
Trying [redacted] 132...
Connected to 210.77.19.132.
Escape character is '^]'.


User Access Verification

Password:
% Password:  timeout expired!
Password:
% Password:  timeout expired!
Password:
% Password:  timeout expired!
```

```
Router#show proc cpu
CPU utilization for five seconds: 65%/19%; o
 PID Runtime(uS)     Invoked      uSecs   5S
   1        4000          21        190  0.0
   2        4000        5146          0  0.0
   3           0       25105          0  0.0
   4           0           1          0  0.0
   5     7220000        3486       2071  0.0
   6           0         515          0  0.0
```

Fig. 15. Mild-DoS: CPU utilization (left) and test with telnet (right)

6.2 Comparison with Related Tools

In this section, we compare RPFuzzer with Peach, SPIKE and Sulley from generation strategy, monitor, debugger, checksum validation, dimensions, the number of test cases, run time, vulnerabilities on the SNMP protocol and so on. The comparison results are shown in **Table 10**.



[redacted]
2[redacted]2
Location:China
Packets Sent:102
Packets Received:92
Packets Lost:10(9% loss)
Response time Minimum:1ms
Response time Maximum:117ms
Response time Average:40ms

```
[2012-4-13 17:44:39]Reply from [redacted] 122: bytes = 64 time = 15ms TTL = 255
[2012-4-13 17:44:41]Reply from [redacted] 122: bytes = 64 time = 2ms TTL = 255
[2012-4-13 17:44:42]Reply from [redacted] 122: Request timed out.
[2012-4-13 17:44:44]Reply from [redacted] 122: bytes = 64 time = 79ms TTL = 255
[2012-4-13 17:44:46]Reply from [redacted] 122: bytes = 64 time = 29ms TTL = 255
[2012-4-13 17:44:47]Reply from [redacted] 122: bytes = 64 time = 37ms TTL = 255
[2012-4-13 17:44:49]Reply from [redacted] 122: bytes = 64 time = 89ms TTL = 255
[2012-4-13 17:44:50]Reply from [redacted] 122: bytes = 64 time = 35ms TTL = 255
[2012-4-13 17:44:52]Reply from [redacted] 122: bytes = 64 time = 86ms TTL = 255
[2012-4-13 17:44:53]Reply from [redacted] 122: Request timed out.
[2012-4-13 17:44:55]Reply from [redacted] 122: bytes = 64 time = 46ms TTL = 255
[2012-4-13 17:44:57]Reply from [redacted] 122: bytes = 64 time = 6ms TTL = 255
```

Fig. 16. Packet Loss Rate and Response Time

Table 10. Comparison results

Name	Strategy	Monitor	Debugger	Dimension	Number	Run time	Speed	Bugs
SPIKE	generation	No	No	Single	62913	95m	662	0
Peach	m&g	Yes	No	Single	76593	224m	341	0
Sulley	generation	Yes	No	Single	52396	868m	60	0
RPFuzzer	m&g	Yes	Yes	Multiple	2153000	1440m	1495	8

- Stragedy. RPFuzzer and Peach which adopt the combination of generation and mutation (m&g) is superior to SPIKE and Sulley in terms of strategy to generating test cases.
- Monitor. Although Peach and Sulley have monitors, they are not or partly not suitable for monitoring routers. For example, Sulley have two monitors, network monitor and process monitor. The former monitor network com13. TIIS-RP-2013-May-0427.R1munication by sniffing NIC devices, which is as inefficient as a sniffer. The latter that monitor related target processes is aimed at applications. Only RPFuzzer' monitor is developed specially for routers.
- Debugger. RPFuzzer has a debugger that can record register values when an exception occurs. Other tools all lack a debugger.
- Run Time, number and vulnerabilities. Because of lack monitor and debugger, the run time of SPIKE is less than that of other tools. RPFuzzer is the most time-consuming for the number of test cases is the most. In addition, the monitor and the debugger also consume a lot of time. To explain the efficiency of RPFuzzer, we compute the overall speed with the formula: speed=number/runtime. The Speed of RPFuzzer is the fastest. Considering a combination of number of test cases, run time and vulnerabilities, RPFuzzer performs more effectively.

7. Conclusion and Future Work

The paper designs the first semi-automatic vulnerability discovering framework of router protocols and proposes a mathematical model TFTCG. Based on above architecture and model, we develop a tool RPFuzzer. RPFuzzer improves test case generation strategy, and solves the problem that the monitors and debuggers of previous tools are not applicable to routers. RPFuzzer can test effectively and automatically test routers, and can be easily extended to test other network devices such as switches. Furthermore, RPFuzzer offers a test case library for testing network applications and other software that involve router protocols.

In the future, we plan to extend our approach in different directions. First, we intend to extend the application scope of the debugger. Second, we want to explore the solutions to the problem about input combination explosion.

References

[1] F. Linder, "Routing and tunneling protocol attacks," in *Proc. of BlackHat briefings*, Amsterdam, Holland, November, 2001. Article (CrossRef Link).

[2] M. Lynn, "The holy grail: Cisco IOS shellcode and exploitation techniques," in *Proc. of BlackHat*, Las Vegas, USA. July, 2005. Article (CrossRef Link).

[3] A. Pilosov and T. Kapela, "Stealing the internet: An internet-scale man in the middle attack," in *Defcon 16*, Las Vegas, USA, August, 2008. Article (CrossRef Link).

[4] Gyan Chawdhary and Varun Uppal, "Cisco IOS shellcode," in *Proc. of BlackHat*, Las Vecas, USA, August, 2008. Article (CrossRef Link).

[5] Groundworks technologies, dynamips gdb server mod project, http://www.groundworkstech.com/projects/dynamipsgdb-mod, June-December, 2011.

[6] National Vulnerability Database, http://nvd.nist.gov/, June-December, 2011.

[7] Felix Lindner, "Cisco vulnerabilities-yesterday, today and tomorrow," in *Proc. of BlackHat*, Virginia, USA, September 29-October 2, 2007. Article (CrossRef Link).

[8] Felix Linder, "Cisco IOS attack and defense the state of art," in *Proc. of 25th Chaos Communication Congress (25C3)*, Berlin, Germany, December, 2009. Article (CrossRef Link).

[9] Felix Linder, "Cisco IOS router exploitation," in *BlackHat*, Las Vecas, USA, July, 2009.Article (CrossRef Link).

[10] A. Cui, J. Kataria and S.J. Stolfo, "Killing the myth of Cisco IOS diversity," in *Proc. of USENIX Worshop on Offensive Technologies*, San Francisco, CA, USA, August, 2011. Article (CrossRef Link).

[11] S. Muniz and A. Ortega, "Fuzzing and debugging Cisco IOS," in *Proc. of BlackHat*, Barcelona, Spain, March, 2011. Article (CrossRef Link).

[12] B.P. Miller, L. Fredriksen and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, 33(12):32–44, 1990. Article (CrossRef Link).

[13] P. Oehlert, "Violating assumptions with fuzzing," *Security & Privacy*, IEEE, 3(2):58–62, 2005. Article (CrossRef Link).

[14] Ai-Fen Sui, Wen Tang, Jian Jun Hu and Ming Zhu Li, "An effective fuzz input generation method for protocol testing," in *Proc. of IEEE 13th International Conference on Communication Technology (ICCT)*, pages 728–731, IEEE, September, 2011. Article (CrossRef Link).

[15] X. Zhu, Z. Wu and J.W. Atwood. "A new fuzzing method using multi data samples combination," *Journal of Computers*, 6(5):881–888, 2011. Article (CrossRef Link).

[16] Z. Wu, J.W. Atwood and X. Zhu, "A new fuzzing technique for software vulnerability mining," in *Proc. of the IEEE CONSEG*, Chennai, India, December, 2009. Article (CrossRef Link).

[17] SPIKE, http://www.immunityinc.com/resourcesfreesoftware.shtml, June, 2010-November, 2011.

[18] PEACH, http://peachfuzzer.com/, June, 2010-November, 2011.

[19] Sulley, http://code.google.com/p/sulley/, June, 2010-November, 2011.

[20] AutoDafe, http://autodafe.sourceforge.net/, June, 2010-November, 2011.

[21] GPF, http://www.vdalabs.com/tools/efs gpf.html, June, 2010-November, 2011.

[22] M. Sutton, A. Greene and P. Amini, *Fuzzing: brute force vulnerabilty discovery*, 1st Edition, Addison-Wesley Professional, New Jersey, 2007.

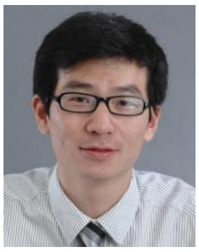
[23] B. ZHANG, C. ZHANG, and Y. XU, "Network protocol vulnerability discovery based on fuzzy testing," *Journal of Tsinghua University (Science and Technology)*, pages S2, 51–56, 2009. Article (CrossRef Link).

[24] G. Banks, M.Cova, V.Felmetsger, K.Almeroth, R.Kemmerer and G.Vigna, "Snooze: toward a stateful network protocol fuzzer," *Information Security*, pages 343–358, 2006. Article (CrossRef Link).

[25] Common Vulnerabilities and Exposures, http://cve.mitre.org/, June-December, 2011.
[26] Qixu Liu and Yuqing Zhang, "TFTP vulnerability finding technique based on fuzzing," *Computer Communications*, 31(14):3420–3426, 2008. Article (CrossRef Link).
[27] GDB, The GNU Project Debugger, http://sources.redhat.com/gdb/, June-December, 2011.
[28] IDA, http://www.hexrays.com/products/ida/index.shtml, June-December, 2011.
[29] J.Case, M.Fedor, M.Schoffstall and J.Davin, RFC 1157: A Simple Network Management Protocol (SNMP), 1990.
[30] SNMPv2 Working Group et al, RFC 1902: Structure of management information for version 2 of the simple network management protocol (SNMPv2), 1996.
[31] R.Mundy, D.Partain and B.Stewart, "Introduction to SNMPv3," *Technical report*, RFC 2570, April, 1999.
[32] O.Tal, S.Knight and T.Dean, "Syntax-based vulnerability testing of frame-based network protocols," in *Proc. of 2nd Annual Conference on Privacy, Security and Trust*, pages 155–160. Citeseer, 2004. Article (CrossRef Link).

Zhqiang Wang is currently a Ph.D. candidate in Department of Communication Engineering, Xidian University, China. He has joined in State Key Laboratory of Integrated Services Networks, in Xidian University. His research interests include system security and network security.

Yuqing Zhang is a professor and supervisor of Ph.D. candidates of Graduate University of Chinese Academy of Sciences. He received his B.S. and M.S. degree in computer science from Xidian University, China, in 1987 and 1990 respectively. He received his Ph.D. degree in Cryptography from Xidian University in 2000. He is a member of IEEE Communications Society and IEICE Transactions on Communications. He has published lots of papers in International Journals and conferences including IEEE Transactions on Dependable and Secure Computing, IEEE Transactions on Wireless Communications, IEEE Communications Letters, Globecom, RAID and so on. His research interests include cryptography, information security and network protocol security.



Qixu Liu is a Post Doctor in University of Chinese Academy of Sciences, Beijing, China. He received his Ph.D. degree in Information Security from Graduate University of Chinese Academy of Sciences, in 2011. He received his B.Sc. in Information Security from University of Science and Technology of China, in 2006. He has worked in network and system security and his current research interests include vulnerability assessment and web security.