PNFUZZ: A STATEFUL NETWORK PROTOCOL FUZZING APPROACH BASED ON PACKET CLUSTERING

HuiHui He and YongJun Wang

College of Computer Science, National University of Defense Technology ChangSha, China

ABSTRACT

Due to the interactivity of stateful network protocol, network protocol fuzzing has higher blindness and lower testcase validity. The existing blackbox-based fuzzing has the disadvantages of high randomness and blindness. The manual description of protocol specification which requires more expert knowledge, is tedious and does not support the protocol without public document, which limits the effect of current network protocol fuzzer. In this paper, we present PNFUZZ, a fuzzer that adopts the state inference based on packet clustering algorithm and coverage oriented mutation strategy. We train a clustering model through the target protocol packet, and use the model to identify the server's protocol state, thereby optimizing the process of testcase generation. The experimental results show that the proposed approach has a certain improvement in fuzzing effect.

KEYWORDS

Fuzzing, Software Vulnerabilities, Network Protocol, Network Packet Clustering.

1. INTRODUCTION

With the rapid development of the Internet and the Internet of Things, various computer technologies have brought great convenience and improvement to people's lives. At the same time, people have more and more important data stored on the Internet, and their dependence on networks and computers is increasing. However, network attacks, data theft and virus spread are becoming more and more serious. For instance, the famous WannaCry [1] ransomware attack was a cyberattack ,which targeted computers on Windows operating system. It made use of "EternalBlue" exploit, which found a security flaw in Windows' Server Message Block(SMB) protocol, an implementation of the application layer protocol which was mainly used as the communication protocol of Microsoft network. The prevalence of software security vulnerabilities is one of the main reasons for the frequent occurrence of such network security incidents. Network server is more likely to be targeted and attacked by criminals because of its exposed communication ports in the public environment. With the popularization of smart devices and the development of the Internet of Things, smart devices with networking functions, its software and network protocols have sprung up one after another, which has exacerbated the severity and challenges of the network security situation [2]. Therefore, digging out potential undiscovered security vulnerabilities and repairing them play a huge role in ensuring the security of cyberspace.

In order to alleviate the losses caused by software vulnerabilities, vulnerability mining techniques are currently effective technical methods. Vulnerability mining technique based on fuzz testing

David C. Wyld et al. (Eds): CSEA, DMDBS, NSEC, NETWORKS, Fuzzy, NATL, SIGEM - 2020
pp. 61-69, 2020. CS & IT - CSCP 2020 DOI: 10.5121/csit.2020.101805

[3][4] has the characteristics of high degree of automation, less expert knowledge, and lower false positive rate. It constructs a large number of deformed and mutated data to input to the software. The software detection technique determines whether the software has caused errors, abnormalities or even crashes after receiving and processing these malformed data. If these conditions occur, analyzing them to determine whether the vulnerability has been triggered and whether it can be exploited.

This paper is organized as follows: Section 2 introduces the previous work and the advantages and disadvantages of the existing methods. Section 3 introduces the principle and components of PNFUZZ. Section 4 analyses the experimental results. Section 5 gives a conclusion.

2. PREVIOUS WORK

There are two challenges in fuzz testing of network protocol. Firstly, network protocol is a kind of agreement rule of network data communication between client and server. It specifies the type, structure and size of data sent by both sides of communication. In the process of fuzz testing, testcases need to meet the protocol specifications to a certain extent, otherwise they are easy to be discarded. Secondly, many network protocols are stateful, and the client and server need to keep communicating in a session. This kind of session needs to be maintained in fuzz testing, otherwise the two sides of communication will not be able to communicate normally, which will lead to the failure of fuzz testing. Faced with these two challenges, there are several "dumb" and "smart" protocol fuzzer. "Dumb" fuzzer sends randomly generated test cases to server under test, without knowing the network protocol adopted by server under test. Therefore, "dumb" fuzzer is suitable for any protocol and is easy to develop, because it only needs to generate testcases randomly for different servers. However, due to the randomness and blindness of "dumb" fuzzer, the effectiveness of testcases is much lower than that of "smart" fuzzer. ProxyFuzzer [5] is a typical "dumb" fuzzer, which randomly modifies the communication data between the client and the server by acting as a proxy between the client and the server. Although this fuzzer can be used in the client and server that maintain the session state, but randomly generated testcases make the crash probability of server very low.

Due to the limitation of cdumb" fuzzer in network protocol fuzz testing, "smart" fuzzer is more popular at present. It tries to understand the internal implementation and format of network protocol to generate testcases. Generally speaking, "smart" fuzzer can understand the data structure of network packets by constructing protocol Finite State Machine(FSM), referencing protocol specification or data syntax. Therefore, "smart" fuzzer can produce testcases with lower blindness and randomness and higher efficiency. For example, Peach [6], BooFuzz [7] and AFLNET [8] are "smart" fuzzers. Taking BooFuzz to test the SMTP protocol for example, BooFuzz is a kind of "smart" fuzz testing framework. Users can decide which parts of the data need to be modified and which parts cannot be modified according to the specification and characteristics of the target protocol. Before starting fuzzing, we need to provide the protocol specification of SMTP. An interaction process between the client and server of the SMTP protocol is shown in Table 1. The italic text can be changed, while other data cannot be changed. Otherwise, the interaction will be interrupted. So the fixed part of the requested data is not modified by the configuration of BooFuzz, but the part and its length of the changeable data can be modified randomly. Then BooFuzz determines the status of the server according to the server response status code, and run the next test. Therefore, the test efficiency of "smart" fuzzer depends heavily on the understanding of network protocol specification and communication data format syntax. Before fuzz testing, we need to learn and understand the target network protocol specification based on some relevant documents, and then manually build the data structure of communication data. Moreover, we also need to build a protocol FSM or state transition diagram to describe the state transition between the client and the server. Therefore, these methods has

complex preparation, low level of automation, tedious process of manually defining protocol format and does not support undocument protocol.

Table 1. A mailing interaction of SMTP.

State	Cilent Request	Server Response
220	Try to connecting…	220 *xxx.com Anti-spam GT for Coremail System*
250	HELO SMTP.xxx.com	250 *OK*
334	AUTH LOGIN	334 *username*
334	*test@xx.com*	334 *password*
235	*123456*	235 *auth successfully*
250	MAIL FROM: *test@xx.com*	250 *OK*
250	RCPT TO: *test2@xx.com*	250 *OK*
354	DATA	354 *Enter mail, end with "." on a line by itself*
250	*hello mail!.*	250 *Message sent*
221	QUIT	221 *Bye*

In this work, we propose PNFUZZ to deal with the challenges and limitations of the above methods. PNFUZZ adopts state inference based on packet clustering algorithm and coverage-oriented mutation strategy. Firstly, PNFUZZ infers the session state by clustering the response packets instead of manually understanding the protocol specification, and selects the corresponding test case according to the session state. This method is used to solve the maintenance of session state in stateful network protocol fuzz testing. Secondly, PNFUZZ uses the coverage oriented mutation strategy to generate the final testcases. This method mutates the request data, and does not need to manually build the data structure of the target protocol data. This method is used to solve the randomness and blindness of testcase generation.

3. TOOL DESIGN AND IMPLEMENTATION

3.1. Description and definition of problems

Before introducing PNFUZZ in detail, we need to introduce some definitions and terms.

Definition 1 (Fuzz Testing). A process of repeatedly inputting testcase into server under test.

Definition 2 (Seed). Test data before mutation.

Definition 3 (Testcase). Mutated seed, which is finally sent to server under test.

Definition 4 (Monitor). In the process of fuzz testing, checking whether server under test generates an exception or crash.

Network protocol is an agreement for communication between programs. According to International Organization for Standardization (ISO), computer network protocols can be divided into five layers: physical layer, data link layer, network layer, transport layer and application layer. Depending on the application of network programs, there are a large number of different application layer protocols, such as File Transfer Protocol (FTP), Domain Name Resolution Protocol (DNS), and Hypertext Transfer Protocol (HTTP). In this paper, the fuzz testing for network protocol mainly refers to fuzz application layer protocol. The network protocol can be classified to stateful protocol and stateless protocol depending on whether the request and

response data are independent. For the stateless protocol, there is no dependence between the request data sent by the client one after the other, and the server only responds to the received client data, and does not depend on the previous request and response data, such as HTTP. For the stateful protocol, the request data sent by the client is determined by the last response data received by the server, such as FTP. Generally, the fuzz testing for file processing software only needs to input each testcase to program under test, and monitor the running and execution of it. However, it is more difficult to fuzz network server. Basing on understanding the data format of network protocol, the testcases can be sent to the server after establishing a connection with the server through the network and fuzzer needs to maintain a session with the server.

3.2. Packet clustering algorithm

Clustering [9] is an unsupervised machine learning algorithm. It divides unlabeled datasets into several categories by similarity metric, and each data is classified into a certain cluster. The data of the same cluster have higher similarity, while the data of different clusters have lower similarity. The popular clustering algorithms are K-means, DBSCAN and Hierarchical Clustering. In this work, we use Aggregate Hierarchical Clustering [10] to divide the response datasets into multiple categories. The advantage of this algorithm is that it does not need to manually specify the number of final clusters. The similarity between datasets can be calculated by distance metric, such as Euclidean distance, Cosine distance, and Longest Common Subsequence Distance. Among them, Longest Common Subsequence Distance [11] is more suitable for the calculation of the similarity of binary data.

Given data *a* and *b*, we define the Longest Common Subsequence Distance *D* as follows:

$$D(a,b) = 1 - \frac{length(LCSS(a,b))}{max(length(a),length(b))}. \quad (1)$$

where *LCSS* is the longest common subsequence algorithm.

3.3. Model and Algorithm

The overall architecture and implementation of PNFUZZ is shown in Figure 1. PNFUZZ consists of data preprocessing, instrumentation, seed selector, state recognition, mutator and two communication channels. We elaborate on each below.

(1) Two Communication Channels

In order to implement the fuzz testing of network protocol, the basic ability of PNFUZZ is to communicate with server under test. PNFUZZ realizes two communication channels. One realizes network communication between PNFUZZ and server under test through socket APIs. The other is to feed back the code coverage information of server under test to PNFUZZ through anonymous pipes and shared memory. It is worth mentioning that instrumentation can insert anonymous pipes and shared memory related code into the server under test.

(2) Data Collection and Processing.

Data collection is the first preparation after selecting the target protocol. We can capture the request and response data between the client and the server through the network sniffing tool such as Wireshark [12] and Tcpdump [13]. In this work, we select Wireshark to capture protocol data. The advantage of Wireshark is that it supports multiple platforms and can extract protocol

session easily with GUI interface. The protocol data captured by Wireshark can be saved to PCAP file for subsequent analysis and processing.

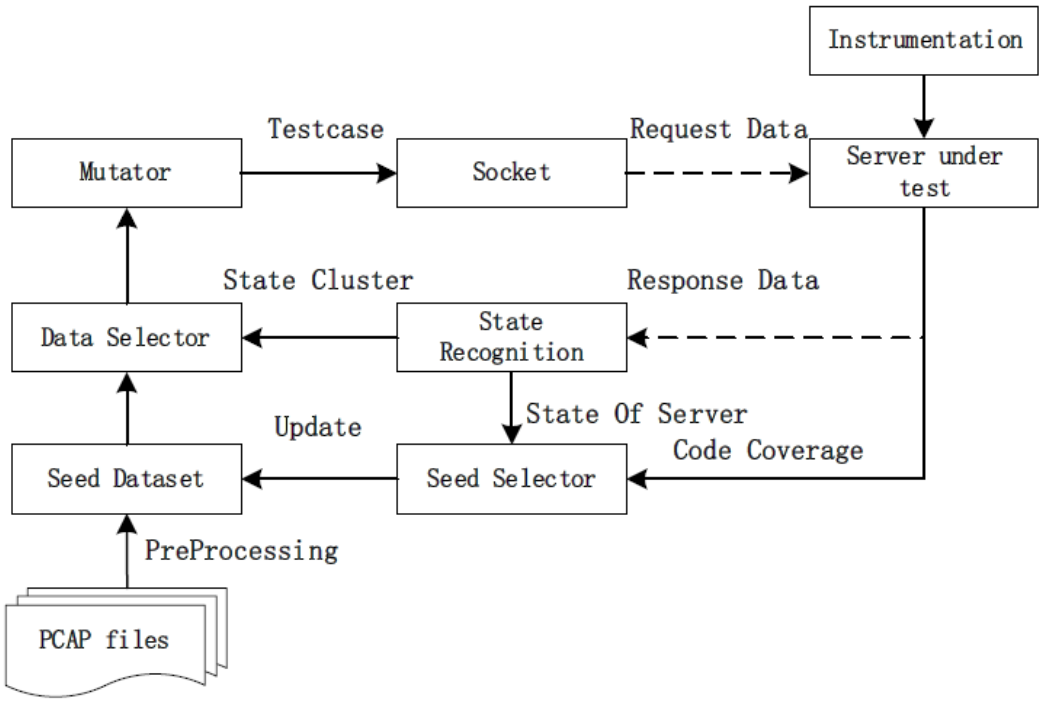

Figure 1. Architecture and Implementation of PNFUZZ

After collecting enough data, it is necessary to preprocess datasets. On the one hand, there is noise in the captured data, on the other hand, PCAP file cannot be directly applied in the fuzz testing. Data preprocessing includes the following steps:

- Removing network packets from non-target protocols.
- Removing invalid and redundant packets due to retransmission mechanism.
- The packets of the session between the client and the server are extracted completely.
- Filling all response packets with "\x00" until the length is the same as the length of the longest packet.
- Establishing relationships between request data and response data for each session.

Finally, we formally describe the processed dataset as follows:

Define request data or seed as $Req = (cid, sid, payload)$, and define response data as $Res = (sid, cid, payload)$.Where cid denote identification of single request data in the request dataset, sid denote identification of single request data in the request dataset, and $payload$ denote binary request or response data. Define request datasets as $REQ = \{Req1, Req2, \dots, Reqm\}$ and define response datasets as $RES = \{Res1, Res2, \dots, Resn\}$

.

(3) Instrumentation

Instrumentation is a technique for executing some extra code during the normal execution of a program. In the aspect of fuzz testing, instrumentation has two important functions. Firstly, it can collect program runtime information, output debugging information and track program execution status. The second is to enhance the sensitivity of the program to anomalies and detect anomalies in time. For example, when some abnormal behaviours occur such as array index out of range

and corrupting local variables on the stack, the program may not crash immediately, but will affect the subsequent program behaviour, even the program will not be affected at all. After instrumentation, the program will crash immediately when it detects abnormal behaviour, and we can check the thread context and core dump in time.

(4) State Recognition

PNFUZZ infers the session state by the response message of serve under test. Before fuzz testing, we need to train a clustering model based on response datasets RES. In this work, we implement Hierarchical Clustering Algorithm by the Agglomerative Clustering component of Sklearn [14] which is a popular machine learning algorithm library. We get the clustering results $C=\{C1,...,Ck\}$ of all session states of server under test and a clustering model M. Each session state cluster Ci contains some response data: $Ci \subseteq RES$. Clusters are disjoint: $Ci \cap Cj = \emptyset$,$\bigcup i=1kCi \subseteq RES$ $(i \neq j, i \leq k; j \leq k)$. In the fuzz testing, the protocol data P of the response packet is extracted, and then the session state cluster Cr of the protocol data is predicted by the trained clustering model.

(5) Seed Selector

According to the result of the state recognition, seed selector randomly selects a seed $Reqr$ from request datasets REQ associated with the session state Cr of server under test and pass it to the data mutator. In addition, if the code coverage $covr$ of server under test is higher than the average code coverage $co\bar{v}$ through anonymous pipes and shared memory or server under test crash, the previous sent testcase Tr' should be added to the request datasets of corresponding states. Otherwise, this testcase will be discarded. Finally, the code coverage $covr$ is recorded into set $COV=\{cov1,...,covk\}$The complete algorithm of state recognition and seed selection is shown in algorithm 1. seed set updating algorithm is shown in algorithm 2.

Algorithm 1 State recognition by packet cluster

Input: RES, REQ ,clustering model M and the target protocol data P

Output: the session state cluster C_r and testcase T_r

$Mat \leftarrow \emptyset$
$cid \leftarrow 0$
for Res_a in RES **do**
    for Res_b in RES **do**
        $Mat \leftarrow Mat \cup D(Res_a, Res_b)$;
    end for
end for
$M \leftarrow$AgglomerativeClustering(RES, Mat)
$C_r \leftarrow$Predict(M, Mat, P)
$C \leftarrow$ RandomSelect(C_r)
$cid \leftarrow$GetMember(C, "cid")
$T_r \leftarrow REQ[cid]$
return C_r, T_r

Algorithm 2 seed set updating

Input: REQ, C_r, T'_r, cov_r and COV

Output: REQ and COV

1: $\overline{cov} \leftarrow$ Average(COV)
2: **if** ($cov_r > \overline{cov}$) | (Server under test crashed) **then**
3: $REQ \leftarrow REQ \cup T'_r$
4: $COV \leftarrow COV \cup cov_r$
5: $\overline{cov} \leftarrow$ Average(COV)
6: **else**
7: $REQ \leftarrow REQ \setminus \{C_r\}$
8: **end if**
9: **return** REQ , COV

(6) Mutator

The mutator mutates the seed selected by the seed selector to produce the final testcase, which is then sent to server under test through the socket network communication channel. PNFUZZ's mutation strategy is based on AFL [15], which has very excellent extension such as LearnAFL[16].

4. EXPERIMENT AND RESULT ANALYSIS

4.1. Experiment Setting

This experiment is based on Ubuntu 18.04 operating system and will be compared with AFLNET and BooFuzz. AFLNET is a state-oriented fuzzing tool for stateful network protocols, which can perform better fuzzing tests on common network protocols. BooFuzz is a stateful blackbox fuzzing framework with high scalability. The FTP protocol server LightFTP [17] and the RSTP protocol server Live555 [18] are the test targets. Using BooFuzz as a benchmark and comparing performance of PNFUZZ, AFLNET and BooFuzz in three indicators: fuzzing efficiency, code branch coverage and vulnerability mining capabilities.

4.2. Result Analysis

The higher code coverage means that the server under test to enter deeper code. Covering more code means more likely to trigger more bugs or vulnerabilities. The code coverage in Table 2 is based on BooFuzz and shows that PNFUZZ and AFLNET have improvement in these two protocols. The code coverage of AFLNET is higher than that of PNFUZZ on LightFTP and lower on Live555. The reason is that AFLNET will make certain adaptations to FTP, and the FTP data format is more simple, while the RTSP data format is more complex. Since BooFuzz is a blackbox fuzzer, its code coverage is inferior to that of coverage-oriented graybox fuzzing. Table 3 and Table 4 show that BooFuzz relies on manual construction of protocol specifications, is faster in the testing process, and about 10 times faster than AFLNET, but the quality of its testcases is relatively poor. So it is difficult to trigger software crash. AFLNET sends testcases at a little bit faster rate than PNFUZZ. The reason is that PNFUZZ needs to recognize server's state through a trained clustering model before sending testcase, which consumes time. Although a certain amount of time is sacrificed, the quality of the generated testcases is better, and in the final the number of crashes triggered is bigger than AFLNET. Table 4 shows that LightFTP did

not trigger any crashes. This phenomenon may be caused by the short fuzz testing time. In short, although the test speed of PNFUZZ is relatively slower, the quality of the generated testcases is relatively better and PNFUZZ is more highly automated and achieve the expected effect.

Table 2. Code coverage improvement.

	LightFTP	**Live555**
AFLNET	54 %	63%
PNFUZZ	51%	68%

Table 3. Sending testcase speed.

	LightFTP	**Live555**
BooFuzz	100~110 count/s	100~110 count/s
AFLNET	9 count/s	9 count/s
PNFUZZ	8 count/s	7 count/s

Table 4. Average number of crashes per hour.

	LightFTP	**Live555**
BooFuzz	0	2
AFLNET	0	6
PNFUZZ	0	7

5. CONCLUSION AND FUTURE WORK

This paper proposes PNFUZZ, a stateful network protocol fuzzer based on the packet clustering algorithm. The trained model of network packet clustering is used to identify the session state of server under test, thereby optimizing the process of selecting testcases. Combining with coverage-guided greybox fuzzing strategy, PNFUZZ improves the quality of testcases.

In the future, more effective protocol reverse techniques and machine learning algorithms can be used to improve the overall fuzzing effect.

ACKNOWLEDGEMENTS

This work was funded by the National Natural Science Foundation of China, and the project approval number was 61472439.

REFERENCES

[1] Akbanov, M., Vassilakis, V. G., & Logothetis, M. D. (2019). WannaCry ransomware: Analysis of infection, persistence, recovery prevention and propagation mechanisms. Journal of Telecommunications and Information Technology.

[2] Khan, M. A., & Salah, K. (2018). IoT security: Review, blockchain solutions, and open challenges. Future Generation Computer Systems, 82, 395-411.

[3] Liu, B., Shi, L., Cai, Z., & Li, M. (2012, November). Software vulnerability discovery techniques: A survey. In 2012 fourth international conference on multimedia information networking and security (pp. 152-156). IEEE.

[4] Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., & Woo, M. (2019). The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering.
[5] Website, "ProxyFuzz," https://github.com/SECFORCE/proxyfuzz/blob/master/proxyfuzz.py , 2020, accessed: 2020-06-17.
[6] Wang, E., Wang, B., Xie, W., Wang, Z., Luo, Z., & Yue, T. (2020). EWVHunter: Grey-Box Fuzzing with Knowledge Guide on Embedded Web Front-Ends. Applied Sciences, 10(11), 4015.
[7] Website, "BooFuzz: A fork and successor of the sulley fuzzing framework," https://github.com/jtpereyda/boofuzz, 2020, accessed: 2020-06-17.
[8] Pham, V. T., Böhme, M., & Roychoudhury, A. (2020). AFLNET: A Greybox Fuzzer for Network Protocols. In Proc. IEEE International Conference on Software Testing, Verification and Validation (Testing Tools Track).
[9] Berkhin, P. (2006). A survey of clustering data mining techniques. In Grouping multidimensional data (pp. 25-71). Springer, Berlin, Heidelberg.
[10] Murtagh, F., & Contreras, P. (2012). Algorithms for hierarchical clustering: an overview. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2(1), 86-97.
[11] Bergroth, L., Hakonen, H., & Raita, T. (2000, September). A survey of longest common subsequence algorithms. In Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000 (pp. 39-48). IEEE.
[12] Orebaugh, A., Ramirez, G., & Beale, J. (2006). Wireshark & Ethereal network protocol analyzer toolkit. Elsevier.
[13] Fuentes, F., & Kar, D. C. (2005). Ethereal vs. Tcpdump: a comparative study on packet sniffing tools for educational purpose. Journal of Computing Sciences in Colleges, 20(4), 169-176.
[14] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. the Journal of machine Learning research, 12, 2825-2830.
[15] Website, "American fuzzy lop (afl) fuzzer," http://lcamtuf.coredump. cx/afl/technical_details.txt, 2020, accessed: 2020-06-17.
[16] Yue, T., Tang, Y., Yu, B., Wang, P., & Wang, E. (2019). LearnAFL: Greybox Fuzzing With Knowledge Enhancement. IEEE Access, 7, 117029-117043.
[17] Website,Lightftp server, https://github.com/hfiref0x/LightFTP, 2020, accessed: 2020-06-17.
[18] R. Finlayson, Live555 media server, http://www.live555.com/ mediaServer, 2020, accessed: 2020-06-17.