Scalable Detection of SIP Fuzzing Attacks

Eric Y. Chen Mitsutaka Itoh
NTT Information Sharing Platform Laboratories
{eric.chen, itoh.mitsutaka}@lab.ntt.co.jp

Abstract

The VoIP technology has been increasingly popular and the number of its users has surged in the past years, because of its economical advantage over the traditional PSTN services. As a side effect, various VoIP servers and clients are becoming attractive targets of malicious attacks. This paper outlines the detection rules we have formulated to detect fuzzing attacks, which attempt to crash a VoIP device by sending it invalid SIP messages. This paper also proposes a system architecture that utilizes multi-core processors in order to scale up the performance of detection using these rules.

1. Introduction

The VoIP technology has been increasingly popular and the number of its users has surged in the past years, because of its economical advantage over the traditional PSTN services. However, this trend has also made various VoIP devices attractive targets of various attacks, since these devices are often deployed on the Internet. VoIP Security Alliance (VOIPSA) [1] was launched in 2005 to invite professionals in the related fields to address the security issues surrounding the VoIP technology. A white paper that enumerates most possible threats to VoIP has been made publicly available by VOIPSA.

This paper focuses on attacks that involve sending malformed messages to a SIP device. In particular, we focus on detecting malformed SIP [2] messages, since SIP is rapidly replacing other VoIP signalling protocols and becoming the de facto standard. In this paper, we use the term SIP fuzzing attacks to refer to this type of attacks.

The remainder of this paper is structured as follows. Section 2 briefly introduces SIP fuzzing attacks. Section 3 surveys the related work that attempts to address this problem. Section 4 outlines the rules we use to detect SIP fuzzing attacks. Section 5 describes how we have implemented a prototype using these rules. Section 6 evaluates this prototype in terms of detection effectiveness and processing performance.

2. SIP Fuzzing Attacks

The word fuzzing is conventionally used to refer to a black-box software testing method that provides random data to the inputs of a program in an attempt to fail the program and find bugs [3]. Bugs found using fuzz testing can be exploited by an attacker to crash or hijack the program. SIP fuzzing refers to such technique designed to test programs or devices that implement SIP.

An alarming number of security incidents discovered through SIP fuzzing have been reported over the past years. In 2003, the US-CERT issued a vulnerability note [4] based on a discovery by the Oulu University Secure Programming Group (OUSPG). According to the note, a significant number of SIP implementations contain vulnerabilities that can be triggered by malformed SIP messages. The impacts range from unexpected system behaviour and denial of service to execution of arbitrary code by buffer overflow. Despite of this warning, a number of vulnerabilities caused by malformed SIP messages are still discovered later in products such as Cisco Firewall Services Module [5], Apple Macintosh OS X VideoConference [6], Linksys VoIP router [7] and Asterisk [8].

An attack that exploits such bugs can be launched in two ways. First, the attacker tries to identify the exact implementation of the target device (vendor and version etc) and then send known malformed packets specific to that implementation. Second, the attacker can flood the target with different patterns of malformed packets (live fuzzing) and hope that at least one of them would trigger a bug in the target. The first way is obviously more efficient while the second way is powerful due to its brute-force nature.

978-0-7695-3329-2/08 $25.00 © 2008 IEEE
DOI 10.1109/SECURWARE.2008.11

3. Related Work

Approaches to the SIP fuzzing attack problem can be generalized into two categories: prevention with extensive fuzz testing before product release, and detection of attacks against online devices.

PROTOS [9] and Asteroid [10] are two well-known tools that are freely available for the purposes of fuzz testing. Codenomicon [11] is the commercial version of PROTOS and provides 10 times more test cases. While preventive measures using these tools are perhaps the most effective way in eliminating software bugs exploitable by a fuzzing attacker, not all developers are willing to make the efforts due to the additional costs incurred. Moreover, users can never be certain if the products they use are free from such vulnerabilities.

Geneiatakis et al. [12] emphasised the importance of having a detection mechanism for SIP fuzzing attacks. They explained the effectiveness of signature-based detection and suggested an example of inspecting INVITE messages. However, this literature offers little details about how one should design the detection rules and implement a detection system for fuzzing attacks.

We agree with Geneiatakis's work in general and see the importance of detecting SIP fuzzing attacks. We had previously presented a preliminary report [18] (in Japanese) that describes our work-in-progress toward designing rules to detect fuzzing attacks. This paper outlines in detail the rules we have formulated and demonstrates the comprehensiveness of these rules by testing them against a commercial fuzzing tool used for software testing. This paper also proposes a system architecture that is capable of utilizing modern multi-core processors to scale up the detection performance.

4. Detection Rules

We propose to use the following five categories of rules to detect malformed SIP packets.

- Incorrect Grammar
- Oversized Field Values
- Invalid Message or Field Name
- Redundant or Repetitive Header Field
- Invalid Semantic

In a paper of this length, we are unable to enumeration all rules in each category. Instead, we describe in detail the general objectives of rules in each category.

4.1. Incorrect Grammar

Rules in this category examine the grammatical structure of each SIP message as defined under Section 25 in RFC3261. For example, the Call-ID field is defined as the following in ABNF:

```
Call-ID=("Call-ID"/"i") HCOLON callid
callid = word ["@"word]
```

In other words, a SIP stack should expect a Call ID to be one string or two strings separated by a "@". The following illustrates two possible invalid structures.

```
Valid
Call-ID: a84b4c7
Invalid 1
Call-ID: a84b4c7@
Invalid 2
Call-ID: a84b4c7, a84b4c7, a84b4c7
```

Invalid structures can be easily detected as the grammar of SIP is concisely defined in RFC3261. Detection rules in this category can be derived from the ABNF.

4.2 Oversized Field Values

An overflow can occur in a well-formed SIP message when any field has a larger integer or a longer string than normally expected. The following is a simple example in which the invalid header field value has an oversized SIP version number.

```
Valid
INVITE Bob@biloxi.com SIP/2.0
Invalid
INVITE Bob@biloxi.com SIP/122214.0
```

Exception handling of such messages can be difficult since the upper limit or the maximum length of each header field value is not always clearly defined in RFC3261. We have created rules in this category based on a number of SIP devices we have worked with. However, rules in this category are implementation-specific and should be adjusted accordingly.

4.3 Invalid Message or Field Name

Rules in this category simply look for method names, response code or header field names not defined in RFC3261. These include messages with mismatched response codes. For example:

```
Valid
SIP/2.0 200 OK
Invalid
SIP/2.0 200 Trying
```

Such messages are fairly easy to detect. However, it is commonly known that a number of so-called dialects exist in SIP and vendors sometimes extend SIP by adding proprietary header fields no defined in RFC3261. Rules in this category should be able to incorporate such extensions.

4.4 Missing or Repetitive Header Field

The following five headers must be present in any SIP message: From, To, Via, Call-ID and CSeq. A SIP message that does not contain any of these headers should be considered malformed.

It is also possible to fuzz a SIP device by sending a message with repetitive headers. We have created rules that look for repetitive headers, with the exception for headers such as Contact, Via, Call-Info and Route, since multiple entries of these headers are allowed. However, since the maximum number of entries for these headers is not defined, the upper limit for each SIP device may be implementation-specific and therefore create an opportunity for buffer overflow attack. An attacker may send a SIP message with excess number of these entries to a level that cannot be correctly handled by the recipient. Therefore, we find it necessary to set up thresholds for the number of occurrences of these headers. Similar to rules described in Section 4.2, these are implementation-specific and should be adjusted accordingly.

4.5 Invalid Semantics

Rules in this category detect messages with anomalies in the semantics, even if the messages may be well-formed and syntactically compliant. A typical example is to spoof one's source IP address to 127.0.0.1 (loopback address) in a SIP message in order to trick the receiver. An unknowing receiver that does not check irregularities in message semantics may send replies to 127.0.0.1 (i.e. itself) continuously and eventually over-consumes its own resources. Other values we inspect include time, date, content length and CSeq.

5. Prototype Implementation

We have created a prototype from scratch and designed the architecture in a way that allows us to utilize multi-core processors that have become very common even in low-end computers. We feel that parallel processing is extremely suitable for detection of fuzzing attacks since the detection process is stateless in nature and can process each packet independently. The architecture is illustrated in Figure 1. This prototype currently supports only SIP over UDP.

Modules in a grey box are collectively executed as an individual detection process. For each core/processor detected in the underlying platform, the Parallel Processing Controller launches a detection process, or more precisely, a Packet Hander, which in our design is an entry object that initiates a new process. Modules in each grey box have a similar architecture as Snort [14], a popular open-source IDS (intrusion detection system).

From the underlying libpcap [15] library, a Packet Handler retrieves packets that match some pre-determined filter rule. Each Packet Handler should retrieve a different set of packets in order to effectively distribute the workload. Each packet received is then passed to the Decoder that decodes layer-3 and layer-4 headers and prepares the data in an expedient manner for the detection engine.

The detection engine implements all rules that we have defined in Section 4. It analyzes every SIP message and look for anomalies. The first rule that matches the SIP message triggers an alert. A message that does not match any rule is considered normal and is forwarded if the prototype is deployed as an SBC (session border controller), or discarded if it is deployed as an out-of-line IDS.

The Log component stores all alerts triggered by Detection Engine in any detection process.

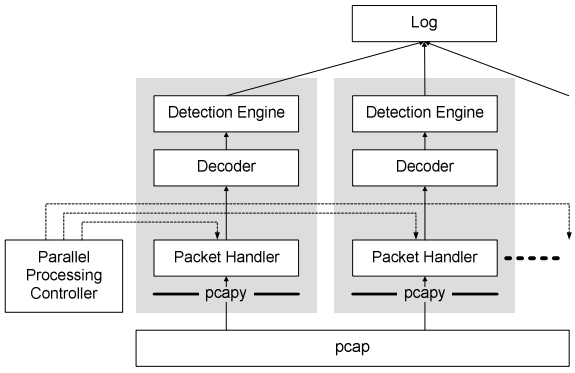

Figure 1. Prototype Implementation

The prototype is currently implemented in Python. We use pcapy [16] as a wrapper for the pcap library. Since the basic Python uses a Global Interpreter Lock (GIL) for thread control that makes utilization of multi-core processors difficult, we implement our Parallel Processing Controller module using Parallel Python [17], which allows us to launch a different process for each detection unit.

6. Experiment

We have conducted experiments to evaluate two aspects of our system – the correctness of the detection rules and the relative performance leverage we can gain by distributing incoming messages to a number of identical processes (instead of threads) running in parallel.

6.1 Evaluation of Detection Correctness

In order to evaluate the correctness of our detection engine, we have used Codenomicon to generate SIP fuzzing attack packets. Although Codenomicon is intended to be used as a fuzzing test tool for legitimate purposes, we found it suitable for our experiment to mimic an attack host. Among the test cases in Codenomicon, we excluded ones that are in the "normal" group as well as ones with anomalies in SDP, which is out of the scope of this paper. As a result, we used a total number of 36398 test cases and feed each SIP message to our prototype. The result is summarized in Figure 2.

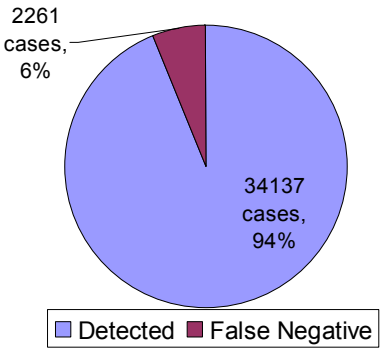

Figure 2. Detection Rate

Using rules in all five categories described in Section 4, we were able to detect anomalies in 34,137 packets, or 94% of the entire test cases selected. In other words, the rate of false negatives is 6% (2,261 undetected packets).

Figure 3 shows the number of matches for each category of rules. Each fuzzing packet may trigger one or more detection rules. The result shows that most test cases generated by Codenomicon contain incorrect grammar and oversized values.

We have also tested our prototype with legitimate SIP messages generated by the following products: OnDO SIP Server, SIPp, X-Lite and Sanyo IP Phone. We also fed our prototype with Codenomicon test cases that are explicitly grouped under the "normal" group. None of these messages trigger any alert and therefore the rate of false positive is 0% with these products.

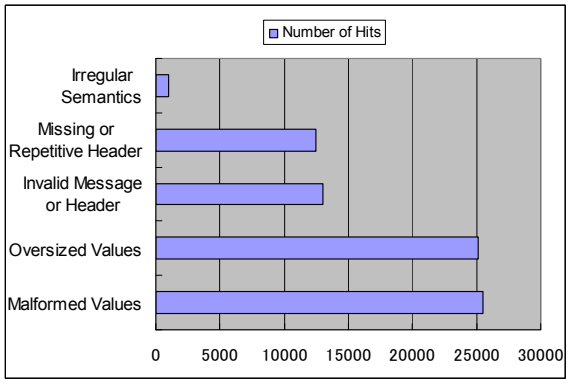

Figure 3. Number of rule matches

After further investigation, we concluded that we were unable to achieve 100% detection rate for two reasons. First, there is still space for improvement in defining our detection rules. After eye-checking some of the undetected packets, we found a large number of them have oversized field values. The problem lies in defining the optimal threshold for each header field value, since thresholds are implementation-specific and require tuning.

We also found a significant number of packets that do not contain any anomaly at all. We have contacted the technical support of Codenomicon and were told that there are indeed a number of valid messages being mixed in the test cases, which sometimes can make fuzz testing more effective. However, since the number of these valid packets is unknown at the time of this writing, we are unable to exclude them from our test result.

6.2. Performance Evaluation

Since our prototype is implemented in Python, we are aware that it is not realistic to expect commercial-grade performance from this implementation. However, we are interested to know if we can leverage multi-core CPUs, which have become very common nowadays, by multiplying our detection engines and execute them in parallel. In our experiment, we have deployed our system in a general PC with the following specifications.

```
OS: OpenSUSE 10.2 x86-64bit
CPU: Intel Core 2 Quad Q6700 @2.66GHz
Memory: 4GB
```

Figure 4 illustrates our test bed. The prototype is deployed out-of-line and inspects packet copies obtained from a tap that sits between a pair of SIPp generators.

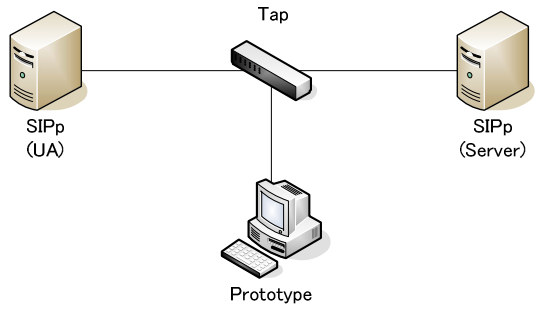

Figure 4. Experiment setup

We decided not to use Codenomicon for this experiment since the size of each packet generated by Codenomicon varies greatly and is not suitable for performance testing. Instead, we use SIPp v2.0.1 as the SIP traffic generator to continuously generate signalling traffic that consist of series of INVITE, OK, RINGING, ACK, BYE, OK messages over UDP at various rates range from 600 to 4800 packets per second (pps). The SIPp-UA machine also generates packets with different source IP addresses to simulate multiple clients. Since our prototype is installed in a machine with 4 cores, we launch different numbers of processes (1 to 4) in each setting and test the maximum throughput. In each experiment setting in which more than 1 process is launched, we hash the IP address and let each process work on an equal portion of SIP traffic.

Figure 5 summarizes the test result. The X-axis represents the rate at which packets are generated by SIPp. The Y-axis represents the number of packets processed by the prototype without packet loss. The dotted line in the graph indicates the ideal scenario in which all packets generated are processed without packet loss. The result from each experiment setting is overlaid on the same graph. With only one process running, the performance starts to degrade at around the rate of 960 pps. As the packet generation rate increase, the result drifts further away from the ideal line. Similarly, the rate at which the performance degrade beings are around 1920, 2880, 3840 pps for experiment settings with 2, 3 and 4 processes.

To our surprise, the performance in each experiment setting is almost perfectly proportional to the number processes launched, although this is expected in theory. At the first attempt, we were able to obtain only 3.5x leverage with 4 processes. After turning off all unnecessary background processes in Linux, we are able to obtain a performance result that is 4 times of the uni-process setting running. We believe this performance leverage is contributed by the fact that all our proposed detection rules are stateless and the workload can be easily distributed to multiple processes as long as a well-balanced hash function can be applied to SIP traffic. This allows us to scale up the performance easily.

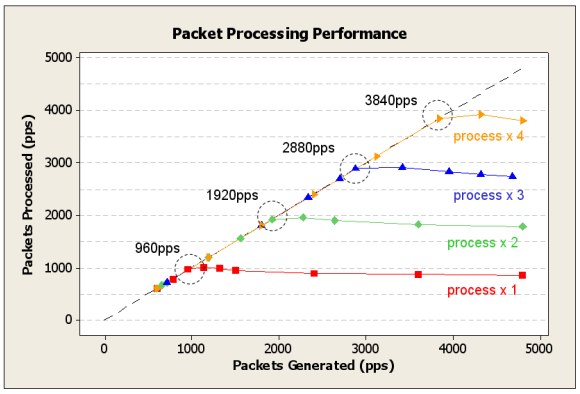

Figure 5. Performance Evaluation

7. Conclusion

Fuzzing attacks pose a serious threat to any SIP device. While good software engineering practices can help reduce potentially exploitable bugs, there will always be devices vulnerable to fuzzing attacks as we have argued. In this paper, we have outlined detection rules effective in detecting fuzzing attacks and have also designed an architecture that scales up the detection process. Our experiment using Codenomicon shows that most known variations of fuzzing attacks can be detected using the five categories of rules we have formulated. We have also demonstrated that the performance can be leveraged by running multiple detection processes in parallel and utilizing multi-core processors.

8. References

[1] VOIPSA, http://www.voipsa.org/

[2] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler. (2002). "SIP: Session Initiation Protocol", RFC3261.

[3] M. Sutton, A. Greene, P. Amini, “Fuzzing, Brute Force Vulnerability Discovery,” Addison Wesley, 2007, ISBN: 0-321-44611-9

[4] US-CERT, “Multiple implementations of the Session Initiation Protocol (SIP) contain multiple types

of vulnerabilities", Vulnerability Note VU#528719, 2003.
[5] US-CERT, "Cisco Firewall Services Module vulnerable to DoS via inspection of malformed SIP messages", Vulnerability Note VU#430969, 2007.
[6] US-CERT, "Apple Macintosh OS X VideoConference SIP heap buffer overflow", Vulnerability Note VU#969969, 2007.
[7] US-CERT, "Linksys RT31P2 VoIP router denial of service vulnerabilities", Vulnerability Note VU#621566, 2006.
[8] US-CERT, "Asterisk null pointer dereference remote pre-authentication DoS vulnerability", Vulnerability Note VU#228032, 2007.
[9] "PROTOS - Security Testing of Protocol Implementations," University of Oulu. http://www.ee.oulu.fi/research/ouspg/protos.
[10] "Asteroid – SIP Denial of Service Tool", http://www.infiltrated.net/asteroid/
[11] Codenomicon, http://www.codenomicon.com/
[12] D. Geneiatakis, G. Kambourakis, T. Dagiuklas, C. Lambrinoudakis, S. Gritzalis "A Framework for Detecting Malformed Messages in SIP Networks", 2005 IEEE Workshop on Local and Metropolitan Area Networks.
[13] D. Geneiatakis, T. Dagiuklas, C. Lambrinoudakis, G. Kambourakis and S. Gritzalis, "Novel Protecting Mechanism for SIP-Based Infrastructure against Malformed Message Attacks: Performance Evaluation Study," CSNDSP 2006.
[14] Snort, http://www.snort.org/
[15] Libpcap, http://www.tcpdump.org/
[16] Pcapy, http://oss.coresecurity.com/
[17] Parallel Python, http://www.parallelpython.com/
[18] M. Ozawa, E. Y. Chen, M. Itoh, M. Hatori, "Evaluation of a Detection System against Fuzzing Attacks on the SIP Protocol", ISEC, Technical Report of IEICE, 2007 (in Japanese)