INTERSTATE: A Stateful Protocol Fuzzer for SIP

Thoulfekar Alrahem, Alex Chen, Nick DiGiussepe, Jefferey Gee, Shang-Pin Hsiao, Sean Mattox,
Taejoon Park, Albert Tam, Ian G. Harris
Department of Computer Science
University of California Irvine
Irvine, CA 92697 USA
harris@ics.uci.edu

Marcel Carlsson
FortConsult
Tranevej 16-18
2400 Copenhagen NV Denmark
mc@fortconsult.net

Abstract

We present the INTERSTATE fuzzer to detect security vulnerabilities in VOIP phones which implement Session Initiation Protocol (SIP). INTERSTATE generates an input sequence for a SIP phone which is constructed to reveal common security vulnerabilities. SIP is a stateful protocol so a state machine description of the SIP protocol is used by INTERSTATE to ensure that the entire state space is explored. The input sequence consists of SIP request messages as well as GUI input sequences which are remotely applied to the phone under test. The input sequence is generated to perform a random walk through the state space of the protocol. The application of GUI inputs is essential to ensure that all parts of the state machine can be tested. Faults are injected into SIP messages to trigger common vulnerabilities. INTERSTATE also checks the SIP response messages received from the phone under test against the expected responses described in the state machine. Checking response messages allows for the detection of security bugs whose impact is more subtle than a simple crash. We have used INTERSTATE to identify a previously unknown DoS vulnerability in an existing open source SIP phone. The vulnerability could not have been discovered without exploring multiple paths through the state machine, and applying GUI inputs during the fuzzing process.

1 Introduction

Several factors have combined to dramatically increase the significance of the computer security problem and the need for computer security research. Software security, a subfield of computer security, has attracted significant research attention relatively recently. Software security involves the protection of software which interacts with a network, either directly or indirectly. A great deal of previous research in indentifying security vulnerabilities in code has involved either static analysis or dynamic checking, but both of these techniques have inherent advantages and disadvantages. Static analysis techniques are employed prior to software deployment, so they have no impact on run time system performance. However, static analysis does not have access run time information, so vulnerability detection is imprecise, either resulting in false positives or false negatives. Run time analysis detects security vulnerabilities on-the-fly, as they occur during execution. Run time analysis is more accurate because it can evaluate the entire dynamic system state to detect vulnerability conditions, but the performance impact can be significant for a tightly constrained system.

We investigate the use of *fuzzing* as a method to detect vulnerabilities in networked software. Fuzzing is performed before software deployment, so it has no impact on run time performance as static analysis. Fuzzing is also a dynamic process

so all of the dynamic system state can be used to identify vulnerabilities, so it has the potential to have the accuracy of run time analysis. Fuzzing is fundamentally different from traditional testing techniques because it requires the violation of assumptions about program behavior [19].

We explore vulnerabilities in VOIP phones, specifically those which adhere to the Session Initiation Protocol (SIP) standard. SIP phone applications are of interest because their use is becoming more widespread, and because their security has not been fully explored. The SIP protocol presents a challenge because is it a stateful protocol, unlike many other application-layer protocols which have no state. The INTERSTATE fuzzer explores the state machine of the SIP protocol during the testing process to reveal vulnerabilities associated with obscure control flow paths. Test sequences are generated and transmitted to the SIP phone to force the phone application to explore control paths automatically. The test sequences include SIP messages which are also modified to include faults which may trigger security vulnerabilities.

The INTERSTATE fuzzer controls the GUI of the phone being tested in order enable the activation of all control paths in the phone under test. For example, a large portion of the SIP state space can only be reached if a phone call is accepted. Full exploration of the state machine is only possible if the GUI inputs of the phone can be controlled by the fuzzer. The INTERSTATE fuzzer also checks the response messages and compares them to the expected responses as described in the state machine. Detailed response checking is needed because some vulnerabilities may manifest themselves as errorneous response messages, rather than as a complete crash of the phone.

The unique features of the INTERSTATE fuzzer are as follows:

1. **Automatic exploration of the protocol state machine**
2. **Application of GUI inputs to the phone under test**
3. **Checking of response messages**

We have used the INTERSTATE fuzzer to evaluate the KPhone SIP phone [1] and we have identified a previously unknown vulnerability in that application.

2 Fuzzer System Structure

Figure 1 depicts the interactions between the basic components of the INTERSTATE fuzzer and a SIP phone under test. The SIP phone is assumed to communicate with users through either a graphics user interface (GUI) or a text-based interface. The fuzzer provides all inputs and examines all message outputs. The edges labeled *messages* indicate message sequences being transferred between the testing system and the system under test via a network. The edge labeled *GUI* indicate the transfer of user commands to the SIP phone over a TCP/IP network. Previous research in security testing has only used the testing system to supply messages and had not controlled the GUI of the system under test [12, 14, 24, 3]. The reason that the user interface is not controlled in previous work is that it is assumed that an attacker would not have the ability to control the user interface, so there is no need to do so during security testing. Although it is true that the attacker would usually not have control over the user interface, an attack could be devised which depends on the actions of the user. For example, an attack on a SIP phone might only be successful if the user accepts the phone call. In order to explore such attacks it is necessary to control the user interface during testing.

The testing system has three parts, the *Protocol Description*, the *Test Sequence Generator*, and the *Response Analyzer*. The protocol description is a state machine which describes the behavior of a phone application which adheres to the SIP protocol. The state machine is manually extracted from the protocol specification in the form of a Request for Comments (RFC) [2]. The test sequence generator produces a sequence of messages and GUI events and sends them to the SIP phone under test. The input sequence forces the SIP phone to follow a random walk of the SIP state machine. Response analysis is performed by using the protocol state machine as a reference. The messages received from the SIP phone are compared to the expected responses contained in the state machine. If the received message sequence differs from the expected sequence then a vulnerability is detected.

3 Related Work

Many static analysis techniques have been applied to software security [6]. The simplest static checking approaches scan the source code for text patterns which are potentially problematic, such as well-known insecure functions like *sprintf* and *gets* [21, 23, 13]. These tools are simple and fast but they produce a large number of false positives. A number of techniques

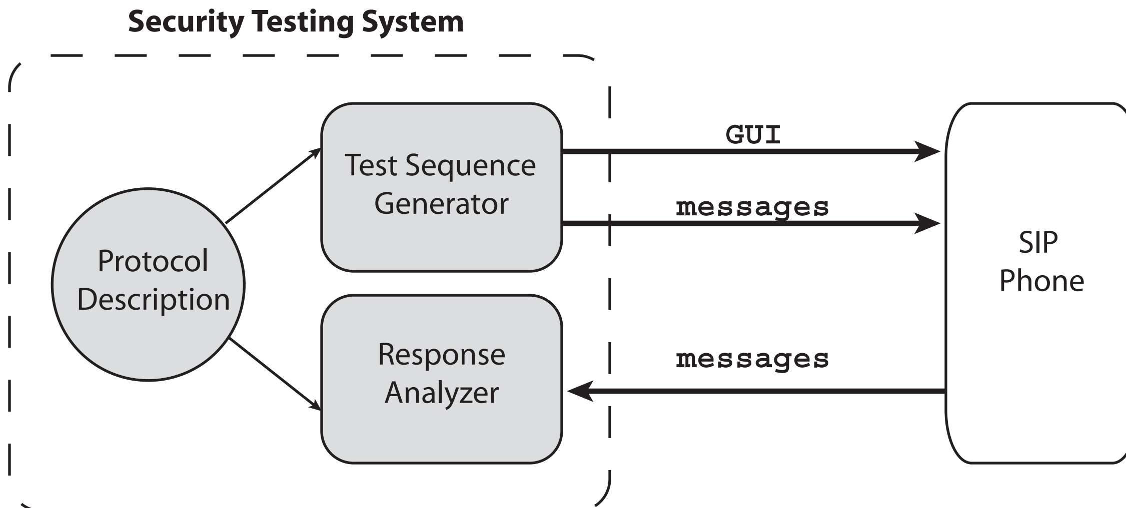

Figure 1. INTERSTATE Fuzzing System

depend on user-specified code annotations to provide functional information about the code which can be verified against. An example is the Splint tool [8, 15] which requires the designer to include preconditions and postconditions for each function. Type modifiers have been used to track the flow of unvalidated data through the system and guarantee that it is not used for a safety-critical operation before it is sanitized [20, 10]. The BOON tool [22] detects buffer overflow vulnerabilities using an integer linear programming formulation to compute the maximum length of buffers on all flow paths.

A number of run time checking techniques to guard against improper manipulation of the stack are based on the approach of StackGuard [7]. The gcc compiler is slightly modified to place and check a *canary* word at the beginning and end of each function call. The canary word is placed on the stack just before the return address and if the return address has been corrupted by a stack smashing attack then the canary word must have been affected. Libsafe [4] and Libverify [5] are dynamic libraries which modify vulnerable functions to include run time checking for stack smashing attacks. Libsafe ensures that a vulnerable function cannot write past the current stack frame. This is accomplished by computing input length and performing a boundary check before each call. *Sandboxing* has been proposed [11] to control a program's access to the operating system by selectively allowing or disallowing the use of particular system calls.

Fuzzing was first proposed by Miller et al at the University of Wisconsin Madison for use in determining program robustness [17, 18, 9, 16]. The approach involves supplying random and directed test sequences to the program under test. The test sequences contain both valid and invalid subsequences which are generated to expose security vulnerabilities. Fuzzing has been used to test implementations of a variety of network protocols including HTTP [12], FTP [14], and SIP [24, 3].

4 Experimental Results

We have evaluated the INTERSTATE fuzzer by using it to test an open source SIP phone, KPhone Version 4.2 [1]. KPhone is written in C++ and C, and totals 45,761 lines if code. The same version of KPhone was tested in previous work [3] and no security vulnerabilities were detected. INTERSTATE detected a timing vulnerability when a Bye message is received immediately after a call is accepted. KPhone loads the required audio codecs before sending the 200 OK message. The process of loading the codecs takes place quickly, usually in less than a second. KPhone crashed when a BYE is received during this codec initialization period.

References

[1] KPhone SIP softphone. http://kphone.cvs.sourceforge.net/kphone/kphone/.

[2] Rfc editor database. http://www.rfc-editor.org/.

[3] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *Proceedings of the 9th Information Security Conference*, 2006.

[4] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks, 1999.

[5] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.

[6] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):32–35, November/December 2004.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Conference*, pages 63–78, Jan 1998.

[8] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, pages 42–51, Jan/Feb 2002.

[9] J.E. Forrester and B.P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *4th USENIX Windows Systems Symposium*, August 2000.

[10] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2002.

[11] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, 1996.

[12] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *International Conference on World Wide Web*, pages 148–159, 2003.

[13] Secure Software Inc. RATS. http://www.securesw.com/rats.

[14] Leon Juranic. Using fuzzing to detect security vulnerabilities. Technical report, Infingo IS, April 2006.

[15] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Usenix Security Symposium*, 2001.

[16] B.P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *International Workshop on Random Testing*, July 2006.

[17] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12), December 1990.

[18] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison, Department of Computer Science, April 1995.

[19] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy Magazine*, 3(2):58–62, March-April 2005.

[20] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[21] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. *ACM Transactions on Information and System Security*, 5(2), 2002.

[22] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[23] David A. Wheeler. Flawfinder. http://www.dwheeler.com/flawfinder.

[24] C. Wieser, M. Laakso, and H. Schulzrinne. Security testing of sip implementations. Technical report, Columbia University, Department of Computer Science, 2003.