The ISC Int'l Journal of Information Security
ISeCure

July 2009, Volume 1, Number 2 (pp. 91–103)
http://www.isecure-journal.org

Security Testing of Session Initiation Protocol Implementations

Ian G. Harris [a,*], Thoulfekar Alrahem [a], Alex Chen [a], Nick DiGiuseppe [a], Jefferey Gee [a], Shang-Pin Hsiao [a], Sean Mattox [a], Taejoon Park [a], Saravanan Selvaraj [a], Albert Tam [a], Marcel Carlsson [b]

[a] *Department of Computer Science University of California Irvine, Irvine, CA 92697 USA.*
[b] *FortConsult A/S, Tranevej 16-18, 2400 Copenhagen NV, Denmark.*

ARTICLE INFO.

Article history:
Received: 12 February 2009
Revised: 11 June 2009
Accepted: 22 July 2009
Published Online: 25 July 2009

Keywords:
Software Security, Voice Over IP, Protocol Fuzzing

ABSTRACT

The mechanisms which enable the vast majority of computer attacks are based on design and programming errors in networked applications. The growing use of voice over IP (VOIP) phone technology makes these phone applications potential targets. We present a tool to perform *security testing* of VOIP applications to identify security vulnerabilities which can be exploited by an attacker. Session Initiation Protocol (SIP) is the widespread standard for establishing and ending VOIP communication sessions. Our tool generates an input sequence for a SIP phone which is designed to reveal security vulnerabilities in the SIP phone application. The input sequence includes SIP messages and external graphical user interface (GUI) events which might contribute to triggering a vulnerability. The input sequence is generated to perform a random walk through the state space of the protocol. The generation of external GUI events is critical to testing a stateful protocol such as SIP because GUI interaction is required to explore a significant portion of the state space. We have used our security testing tool to identify a previously unknown vulnerability in an existing open source SIP phone.

1 Introduction

Several factors have combined to dramatically increase the significance of the computer security problem and the need for computer security research. Our society has become dependent on the use of networked computing devices for all manner of endeavor, even in cost-critical and life-critical applications. At the same time, the number of attacks on computers has skyrocketed. Records kept by the CERT Coordination Center (CERT/CC) show that the number of computer attack incidents reported had increased exponentially until 2003, after which year the data was no longer published [1]. Recently many more insidious large-scale attacks have been launched against private and industrial targets. These include an attack against a MasterCard transaction processing company [2], the posting of search queries from America On Line [3], and the theft of personal information about students and employees at the University of Texas [4] and UCLA [5]. Each of these attacks involved the exposure of personal information for well over 100,000 people in each incident.

The security threat is multiplied by the now com-

* Corresponding author.
Email addresses: harris@ics.uci.edu (I. G. Harris), t@alrahem.net (T. Alrahem), chenat@uci.edu (A. Chen), jdgee@uci.edu (J. Gee), shangpih@uci.edu (S. P. Hsiao), smattox@uci.edu (S. Mattox), taejoonp@uci.edu (T. Park), selvaras@uci.edu (S. Selvaraj), atam@uci.edu (A. Tam).
ISSN: 2008-2045 © 2009 ISC. All rights reserved.

mon use of automated attack tools [6] which allow one individual to quickly launch attacks against thousands of target machines. Automated attack tools have enabled the growth of massive "botnets" which are groups of computers whose operating systems have been completely compromised by rootkits [7]. These botnets, which can control thousands of computers, are leveraged to perform all manner of large-scale network attacks, including simple spam [8], DDoS attacks, click fraud, and port redirection [9].

These types of computer attacks threaten the feasibility of secure communication because any data passing through a compromised machine is not secure. The mechanisms which enable the vast majority of computer attacks are based on design and programming errors in software [10–14]. This observation leads to the conclusion that the threat of computer attacks can be greatly alleviated by improving the software engineering process [15]. Major software manufacturers have appreciated the need in software security [16] but have not been able to effectively address the problem [17].

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. The types of software usually examined include *networked software* which communicates with other computers over a TCP/IP network. These applications communicate using a variety of application-layer protocols such as a web browser/server using HTTP and a voice over IP (VOIP) phone using SIP. Other software which interacts indirectly with a network can be included as well, such as a document viewer which is invoked by a browser to display a downloaded file. This class of networked software and related helper applications is critically important from a security standpoint because it is through this software that attackers can remotely manipulate a computer. Networked software is the interface between a computer and the network, so it is also the gateway through which all attacks must pass. Breaking networked software is an important goal of individuals who intend to compromise the security of a computer system via the network.

The bulk of the previous work in identifying 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 to 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 *security testing* as a method to detect vulnerabilities in communications software. Security testing is performed before software deployment, so it has no impact on run time performance as static analysis. Security testing 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. Security testing is fundamentally different from traditional testing techniques because it requires the violation of assumptions about program behavior [18].

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 it is a stateful protocol, unlike HTTP and some other application-layer protocols which have no state. Our security testing approach 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 selected control paths. The test sequences include SIP messages which are also modified to include faults which may trigger security vulnerabilities. We have applied our security testing tool to evaluate the KPhone SIP phone [19] and we have identified a previously unknown vulnerability in that application.

An innovation of our work is the generation of external user interface events during the security testing process. Since the majority of VOIP phones have graphical user interfaces, our tester generates GUI events. The generation of GUI events is critical to the testing of a stateful interactive protocol such as SIP because GUI events are required to reach a significant portion of the state space. For example, most of the SIP protocol state machine can only be reached if the user at the receiving end accepts an incoming call. This typically requires the generation of a GUI event to mimic clicking an appropriate button. Previous SIP testing approaches which do not apply GUI events [20–24] cannot explore large portions of the state space, and therefore will not identify security vulnerabilities in large portions of code.

The remainder of this paper is organized as follows. The structure of our security testing system is presented in Section 3. Section 4 presents the SIP protocol and the state machine which describes it. Section 5 describes software vulnerabilities which are common to networked applications in general, and which our tool attempts to expose in SIP phones. Section 5 also presents the fault injection functions we use to trigger known vulnerabilities. The algorithm used by our security testing tool to generate tests and check responses is described in Section 6. Section 2 describes related work. Experimental results are presented in Section 7 and the conclusions are discussed in Section 8.

2 Related Work

The following is a brief summary of previous research in areas related to software and application security. Previous efforts are subdivided into four categories. The first category is **Static Analysis** which describes static techniques applied to analyze software before it is deployed. The next category is **Run Time Checking** (also referred to as *dynamic analysis*) which describes approaches which check security properties at run time. Next is **Language-Based Security** which describes the development of new languages (or dialects) to enhance security. The final category is **Testing for Security** which describes approaches that execute a program with test data prior to code deployment.

2.1 Static Analysis

A multitude of static analysis techniques have been applied to software security [25]. 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* [26–28]. These tools are simple and fast but they produce a large number of false positives.

A number of techniques depend on user-specified code annotations to provide functional information about the code which can be verified against. The Splint tool [29, 30] requires the designer to include preconditions and postconditions for each function. The Eau Claire tool [31] translates each C instruction into a *verification condition* which formally describes the impact of the instruction on the program variables. Function annotations and the verification conditions are passed to an automatic theorem prover to check if security conditions are violated. The meta-compilation approach [32] uses system-specific programming extensions written by the user to check code paths for security properties of their design.

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 [33, 34].

The BOON tool [35] detects buffer overflow vulnerabilities using an integer linear programming formulation to compute the maximum length of buffers on all flow paths.

Model checking techniques have been used to verify software security properties in the MOPS static checker[36]. The program is abstracted by considering only "security-relevant operations", meaning those which have an explicit and well defined security impact, such as *chroot* or *seteuid* in UNIX.

2.2 Run Time Checking

A number of techniques to guard against improper manipulation of the stack are based on the approach of StackGuard [37]. 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. By checking the canary word before using the return address, the attack is thwarted. Improvements on this theme include [38, 39]. Compiler extensions have been used to detect other types of attacks as well including format string attacks [40] and TOCTOU attacks [41].

Techniques which focus exclusively on protecting the stack neglect more general buffer overflow attacks which are more difficult to exploit, but are a significant danger nonetheless. Researchers have addressed general buffer overflows in [42] by maintaining an *object table* which stores the location and size of each object in use. They create a run time library for gcc which causes each pointer to be verified against the object table to guarantee legality.

Libsafe [43] and Libverify [44] 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. Libverify improves on Libsafe by adding a *canary stack* where return addresses are pushed in addition to the normal stack. The contents of the canary stack are used to verify the return address popped off of the normal stack.

Sandboxing has been proposed [45] to control a program's access to the operating system by selectively allowing or disallowing the use of particular system calls.

2.3 Language-Based Security

Attempts have been made to modify fundamental aspects of programming languages to support security in a transparent way. Most of these techniques involve a mixture of static and run time methods. For instance, variants of the C language have been defined which are more secure. The CCured type system [46] uses both static type inferencing and run time checks such as bounds checks to ensure safe use of pointers. The Cyclone dialect of C [47] addresses security by eliminating some of the most insecure features of C, provides static restrictions on the use of pointers, and adds some run time checkers as well.

The Java language has several built in security safeguards. An important security aspect of Java is that it does not allow the programmer to directly access and manipulate pointers as C does. This restriction alone goes a long way towards preventing insecure access of memory. The Java virtual machine also implements a stack inspection algorithm [48] which checks the frames in the caller's stack sequence to determine if a process has authority to perform a risky action. Researchers have proposed a static technique for ensuring the same security policy as stack inspection [49].

2.4 Security Testing, Fuzzing

Security Testing, the class of techniques which includes the contributions of this paper, describes approaches that execute a program in order to reveal security vulnerabilities. Testing is performed before deployment of the code like static analysis, so it causes no run time overhead. Security testing approaches use a technique generally referred to as *fuzz testing* [18, 50]. Fuzz testing was first proposed by Miller et al at the University of Wisconsin Madison for use in determining program robustness [51–54]. The approach involves supplying random and directed test sequences to the program under test. These early fuzzing techniques evaluated operating system functions which accept command-line inputs. The functions were executed many times with both valid data, chosen randomly from a known domain, and invalid inputs which were intentionally chosen outside of the valid domain. The functions were *stateless* because the results of one execution did not depend on the results of previous executions. Fuzzing was originally used for robustness testing, looking for erroneous responses that were unrelated to security.

More recently fuzzing has been used to test implementations of a variety of network protocols and to search specifically for security vulnerabilities. The most commonly investigated protocol is HTTP because it is so widely used. One example of many is the testing approach presented in [55] which uses a crawler to identify all "access points" of a site, the forms which accept user input. Random valid data is supplied to the fields of each form, and some of the field values are modified by injecting faults which are known to trigger common vulnerabilities.

A more challenging protocol testing problem involves the testing of *stateful* protocols, those protocols for which the message application sequence has an impact of the responses produced by the protocol implementation. Testing stateful protocols is difficult because the input space which must be considered contains all permutations of valid inputs. Some researchers have examined stateful protocols, including FTP [56], and SIP [20–24]. However, existing approaches are strictly limited in the subset of the state space which is explored during testing. For example, the SIP testing techniques presented in [22] and [23] repeatedly apply only the following message sequence: INVITE, CANCEL, ACK. This restriction ensures that only one dialog exists at a time, but the majority of the state space defined by the SIP protocol cannot be covered. The research presented in [20] applies either sequences of INVITE messages, or ACK messages. The work presented in [21] requires the definition of a manual sequence scenario; their results applied a sequence of INVITE and then CANCEL. The contribution of our work is to allow a much larger subset of the state space to be explored during the testing process.

3 Security Testing System Structure

Figure 1 depicts the interactions between the basic components of our security testing system and a SIP phone under test. The SIP phone is assumed to communicate with the user through a graphics user interface (GUI) which is the most common style of interface found in VOIP phones. The security testing system 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 [22, 23, 55, 56]. 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) [57]. 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.

4 Session Initiation Protocol (SIP)

Session Initiation Protocol (SIP) is used to initiate and terminate communication sessions between VOIP phones. The logical entity which creates a new request, such as initiating a new SIP session, is referred to as a **user agent client** (UAC), and the node receiving the request is a **user agent server** (UAS). A standard phone must have the ability to act as both UAC and UAS depending on whether it is placing a call or receiving one. A UAC and UAS can communicate directly without the intervention of other SIP communication nodes.

SIP defines several different additional types of communication nodes which support phone service by enabling user mobility and several other services. Additional SIP communication nodes include registrar servers which store current user locations, and proxy servers which route requests to a user's current location. To simplify the testing process we assume that a UAC and a UAS are communicating directly, without the assistance of other SIP communication nodes. As a result of this assumption, our testing process can identify security vulnerabilities involving the communication between a UAC and a UAS, but not involving other communication nodes.

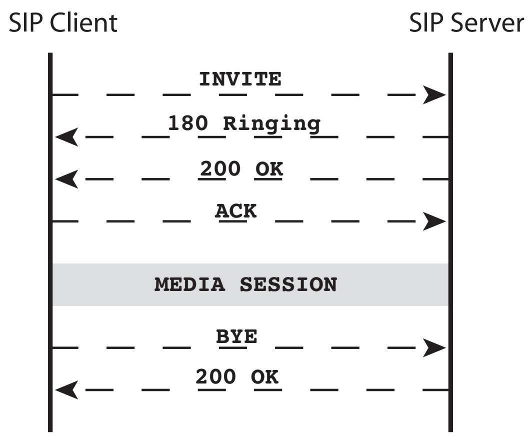


Figure 2. SIP Message Sequence Chart

4.1 SIP State Machine

Our security testing approach requires a state machine description of the protocol to serve as an oracle for response checking, and to bound the test generation process. This state machine must be created manually based on a specification of the protocol. Fortunately, specifications of all public internet protocols are freely available as RFCs [57]. RFCs are available in many natural languages, including English, combined with a set of message sequence charts which describe common communication sequences. The RFC document for the core of this protocol, RFC 3261, describes the set of legal command messages, such as INVITE to start a new session, and the legal responses. In addition, several message sequence charts are presented to describe typical sequences. The chart in Figure 2 shows the creation and completion of a media session between a client placing the phone call and a server receiving the phone call. Using the information in the RFC document it is straightforward to generate a state machine describing the protocol as has been done in previous work [23].

The state machine of a SIP UAS which we used during testing is shown in Figure 3. The behavior of a UAS changes based on the settings specified by the user. Figure 3 describes the behavior of a UAS given “default” settings, where no password is required, “Do Not Disturb” is not selected, and an explicit *offhook* signal from the user is required to accept a phone call. Several transitions are considered during testing which are not shown in Figure 3 for clarity. Each transition is labeled *input/output* where *input* is the input event which triggers the transition, and *output* is the set of output events which are produced as a result of executing the transition.

There are three different classes of transition triggers. The first class of trigger, the receipt of a request message, is labeled **R:***rtype*, where *rtype* is the type of request (i.e. INVITE, ACK, etc.). The transition

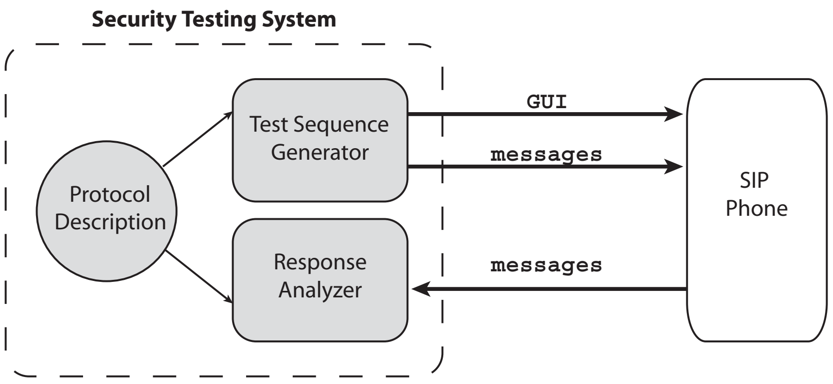


Figure 1. Security Testing System

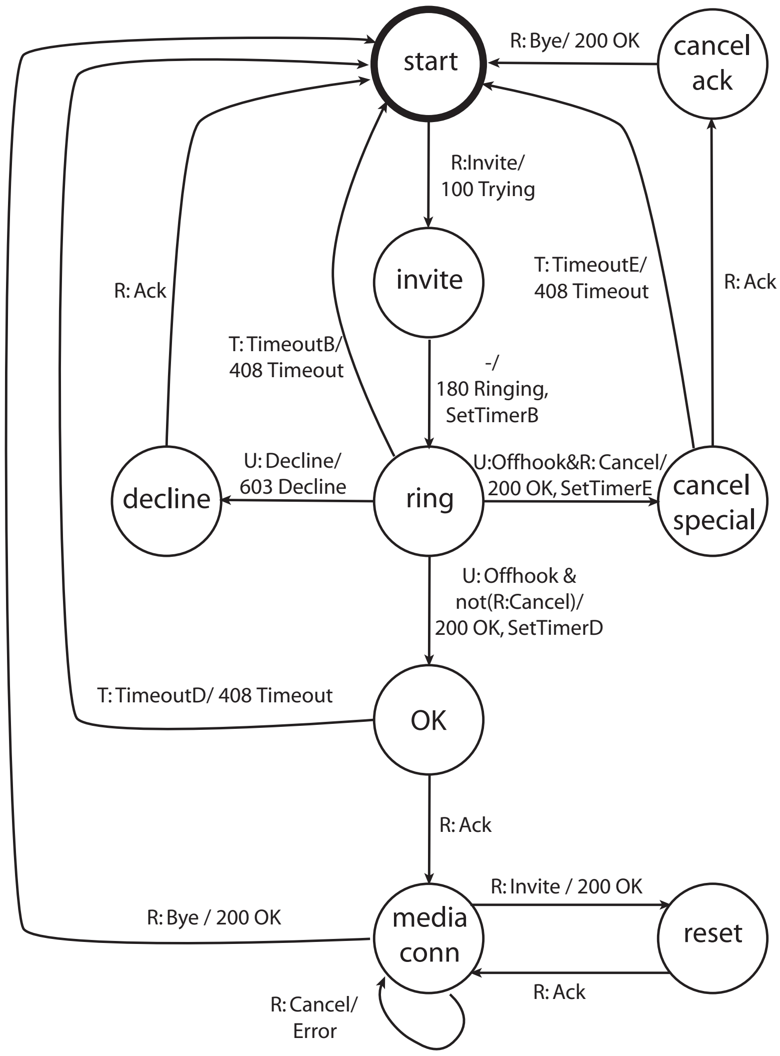


Figure 3. SIP UAS State Machine

between states *start* and *Invite* is triggered by receiving an *Invite* massage.

The second class of trigger is the receipt of user input by the UAS. The types of user input that we are most concerned with for the SIP protocol are the *Offhook* and *Decline* signals generated when the user wants to answer a call or decline a call respectively. User input is labeled **U:***signal* where *signal* is the name of the user input data received. In Figure 3, the transition between states *ring* and *decline* is triggered by the receipt of the *Decline* signal from the user.

The third class of trigger is the expiration of one of the many timers built into the UAS. In Figure 3, the transition between states *OK* and *start* is labeled **T:***TimeoutD*, where *TimeoutD* is the name of the timer whose expiration triggers the edge. Notice that *TimerD* is set at the transition between states *ring* and *OK*.

5 Security Vulnerabilities in Software

It is important to understand the nature of the flaws in software which may manifest themselves as security vulnerabilities in SIP phone applications. Software security vulnerabilities have been studied in previous work [10–14] and have been grouped into a number of broad categories. We present a subset of these vulnerabilities which could exist in SIP phone applications.

- *Unvalidated Input:* This occurs when a program uses external input without checking the properties of the input to ensure that it is legal. An input may be illegal for many reasons including excessive length and the use of illegal characters. This software vulnerability leaves a system open to a number of attacks including buffer overflow attacks and command insertion attacks.
- *Broken Access Control:* This occurs if there is a control path through the program which allows access to an object without checking access permissions properly. This vulnerability can allow direct access to critical data. For a SIP phone critical data might include a password to receive voice mail or change settings remotely.
- *Non-Atomic Check and Use:* Access to an object may be granted based on some aspect of the system state, such as a semaphore indicating that critical data is not exposed. A vulnerability exists if there is a gap in the time between when the system state is checked and the time when the protected object is accessed.

The following is a list of typical attacks which can exploit the above mentioned security vulnerabilities.

- *Buffer Overflow:* A user input is placed directly in a buffer without checking the input length. By creating a very long input the user can exceed the length of the buffer and write into adjacent memory locations. If the buffer is in the heap then the impact of this attack depends on the contents of the memory adjacent to the buffer. If the buffer is a stack frame then the return address can be modified, allowing arbitrary code to be injected and executed.
- *Shell Command Injection:* This occurs when some external input, for example a field of a SIP message, is directly interpreted by a shell. If the external input contains shell commands then an attacker can execute arbitrary shell scripts on the remote machine.
- *Time-of-check Time-of-use (TOCTOU):* This is a race condition that occurs when a file is accessed abstractly by a name which is later resolved to a file handle. A program first checks the access permissions of a file, and then it accesses the file some time later. In the time between the permission check and the use of the file, a malicious user may remap the filename to point to another file (such as a password file) with different permissions.

Each class of attacks depends on the existence of one or more software vulnerabilities. For example, a *buffer overflow* attack almost always depends on the existence of some *unvalidated input.*

5.1 Detection of Security Vulnerabilities

The general approach that we use to detect security vulnerabilities in code is composed of two steps:

(1) Execute the control path containing the vulnerability
(2) Alter the input data to trigger the vulnerability, if necessary.

The first goal of executing the faulty control path is difficult because we have no knowledge of which control path contains the vulnerability. To accomplish this goal we execute a random walk through the protocol state space. The test time required to exhaustively explore a state machine using a random walk can vary widely based on the structure of the state graph. In spite of this fact, the experimental results presented in Section 7 show that testing time for the VOIP phone example which we used is low.

The second goal of altering the input data to trigger the vulnerability is performed by applying a set of *fault injection functions* to modify the contents of message fields. A variety of fault injection functions have been employed in previous work [23]. We use the following subset of these functions which trigger the vulnerabilities that we have described.

- **string_repeat** - This function creates a very large string from a shorter string by repeating the string pattern. Applying this function to a message field can reveal an unvalidated input by mimicking a buffer overflow attack. The long string will overflow its buffer and overwrite other parts of memory, eventually causing a crash or serious malfunction.
- **shell_command_injection** - This function inserts a random shell metacharacter into a random point in a string. Applying this function to a message field can reveal an unvalidated input by mimicking a shell command injection attack. If the metacharacter causes the program to crash, then it is likely that the field is being passed to a shell without validation. This function may reveal broken access control if the injected commands access a protected file such as a password file.

Because we have limited our fuzzer to these two functions, our fuzzer is only likely to detect unvalidated input vulnerabilities using string_repeat and shell_command_injection, and broken access control vulnerabilities using shell_command_injection.

6 Security Testing Algorithm

Figure 4 shows the algorithm used to generate test input and check test responses. The testing tool keeps track of the current state of the UAS in the variable *curr_state* which is initialized on line 1. The algorithm enters a loop between lines 2 and 10 in which tests are generated until either an error is detected or the test process is aborted. Tests are generated by selecting successive edges to traverse and generating their trigger conditions. An outgoing edge from the current state is selected to be traversed on line 3. The algorithm performs a random walk through the state machine, so the selection is random. Once an edge is selected, the triggering conditions of the edge are generated by the *generate_trigger* function which is called on line 4. The response message from the UAS is received on line 5 and checked on line 6. If the response is incorrect then the testing process exits on line because a bug has been found. If the response is correct then the *curr_state* is updated to the successor state of the edge and the new iteration begins.

Figure 5 shows the algorithm used to implement the *generate_trigger* function which is invoked in the security testing algorithm to generate the trigger condition of an edge. The variable t is used to refer to the triggering condition of the edge e. The triggering condition can be one of three types: 'user_input' to represent a GUI event, 'timer' to represent the expiration of a timer, and 'message' to represent the receipt of a message. Each of these three conditions is handled by code in Figure 5 starting at lines 2, 3, and 4, respectively. If the trigger is of type 'user_input' then the X11::GUITest library functions [58] are used to generate the GUI event on the UAS machine. If the trigger is of type 'timer' then the security test process waits for the duration of the timer, plus 1 second to account for network delays. It is sufficient to add only 1 second because the network used during testing is tightly controlled, containing only two machines, one running the security testing tool and the other running the UAS.

```
1.   $curr\_state$ = 'start'
2.   while() {
3.       $e$ = select_outgoing_edge($curr\_state$)
4.       generate_trigger($e$)
5.       $r$ = get_response_message
6.       if (!correct_response($r$)) then
7.           exit (error detected)
8.       else
9.           $curr\_state$ = e.successor_state
10.      }
```

Figure 4. Security Testing Algorithm

If the trigger is of type 'message' then a message must be generated and sent the UAS. A valid message is created first, and then faults may be injected into that message based on the outcome of several random variables. The valid message created on line 5 of Figure 5 is the SIP request class required to trigger the edge, either Invite, Ack, Bye, or Cancel. All fields of the valid message are correctly formatted according to the SIP protocol. The Call-ID field matches that of the current session, unless a new session is being created with an Invite message, in which case a new Call-ID is created. The $fault\_inject\_message()$ function invoked on line 6 generates a random number which is used to determine if a fault will be injected into the message. The probability that the $fault\_inject\_message$ function will return TRUE can be changed in order to control the degree of invalid data received by the UAS. The $fault\_inject\_message()$ function returns TRUE with a 60% probability for this experiment.

If the message is selected for fault injection, a message field for fault injection is selected on line 7 and a fault injection function is selected on line 8. The message field is selected randomly from the following four required fields: From, To, CSeq, and Call-id. The fuzzing function is chosen randomly from between the string_repeat and shell_command_injection functions described in Section 5. The chosen fault injection function is applied to the appropriate message field and the message is sent to the UAS.

generate_trigger(e)

```
1.  $t = e.trigger$
2.  if $t.type$ == 'user_input' then generate_gui_input($t$)
3.  else if $t.type$ == 'timer' then wait($t$) + 1
4.  else if $t.type$ == 'message' then {
5.      $m$ = make_valid_message($t$)
6.      if (fault_inject_message() == TRUE) {
7.          $field$ = select_finj_field()
8.          $funct$ = select_finj_function()
9.          $m$ = apply_finj_function($m$, $field$, $funct$)
10.         }
11.     }
12. send_message($m$)
```

Figure 5. Algorithm to Generate the Trigger Conditions for an Edge

7 Experimental Results

We have evaluated our security testing tool by using it to test an open source SIP phone, KPhone Version 4.2 [19]. KPhone is written in C++ and C, and totals 45,761 lines if code. The same version of KPhone was tested in previous work [23] and no security vulnerabilities were detected. Our security testing tool was executed on a 1.3 GHz AMD Athlon processor with 512MB RAM, running Debian Linux. KPhone was executed on 1.3 GHz Intel Celeron M processor also running Debian Linux.

Our security testing tool detected a timing vulnerability when the UAS traversed the edge between the *ring* state and the *OK* state as shown in Figure 3. When the UAS is in the *ring* state and the user accepts the call (asserting the $Offhook$ signal), 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. The correct operation of a SIP phone from either the *ring* or *OK* states should have been to ignore the BYE since the session has not been completely established until the *media conn* state has been reached. Although the edges triggered by receiving a BYE message from the *ring* and *OK* states were removed from Figure 3 for clarity, they are part of the state machine used by our tool. To ensure that the vulnerability was not specific to a particular machine or operating system, we manually replicated the KPhone crash by running KPhone on a 2.0 GHz Intel Pentium M processor with 1GB of RAM, running Gentoo Linux.

Iteration 1:
- **Edge 1:** start → invite
- **Edge 2:** invite → ring
- **Edge 3:** ringing → start

Iteration 2:
- **Edge 4:** start → invite
- **Edge 5:** invite → start

Iteration 3:
- **Edge 6:** start → invite
- **Edge 7:** invite → ring
- **Edge 8:** ring → OK (crash)

Figure 6. State Machine Path Explored During Testing to Detect Vulnerability

In order to find this vulnerability our tool was run for approximately 6 seconds real time. During that time, the following sequence of 8 edges shown in Figure 6 was traversed in the UAS state machine before the vulnerability was detected. The sequence is grouped into three iterations, each of which describes the establishment and ending of a session. Note that most of the edges triggered by the receipt of a Cancel message and a Bye message are not shown in Figure 3 for clarity. For this reason, Edge 5 and Edge 8 in the sequence shown in Figure 6 are not shown in Figure 3, although they are used during testing.

7.1 Analysis of Results

The vulnerability detected in KPhone is an unvalidated input vulnerability which enables a Denial of Service (DoS) attack. We have demonstrated that this vulnerability can be triggered remotely by sending a BYE message to the KPhone UAS immediately after the user at the receiving end has accepted the phone call. In order to execute this attack in practice, the BYE message must be timed so that it is received just after the call is accepted. This could be accomplished fairly easily by transmitting a number of BYE messages in rapid succession in order to ensure that at least one message is received in the critical time window after phone call acceptance.

An important aspect of the detection of this vulnerability is that the detection was enabled by the application of GUI events during testing. The vulnerability is exposed when the SIP UAS moves from the *ring* state to the *OK* state. Without the ability to assert the $Offhook$ signal through the GUI, this vulnerability could not have been detected because the corresponding state machine edge could not have been triggered. This vulnerability is an example a class of vulnerabilities which are triggered only in portions of the state machine which are accessible through edges triggered by GUI events.

It is also important to note that this vulnerability was detected due to the *message sequence* rather than the specifics of the message content. Only a fuzzer which has the ability to explore the state space by applying many sequences of message types could have detected this vulnerability. Our fuzzer, and all other network application fuzzers, apply fault injection to corrupt the contents of test messages. In this case, fault injection did not detect the vulnerability. This vulnerability was detected using variation in the message sequence resulting from exploring the state machine, rather than variation in the message content induced by fault injection. This underscores the importance of state space exploration when fuzzing stateful protocols.

8 Conclusions

We have presented a security testing tool which ensures VOIP communication security by identifying vulnerabilities in SIP phones. The technique performs a random walk of the SIP state machine by supplying an input sequence to trigger edges in the state machine. Faults are injected in the input message sequence in order to reveal a set of security vulnerabilities which are data dependent. The testing approach depends on the availability of a SIP state machine which describes the behavior of a SIP server. The manual state machine generation process is tedious but it is only performed once and the state machine can be reused to test any SIP phone. We have used our security testing tool to efficiently identify a previously unknown vulnerability in an existing open source SIP phone. In the future, this technique can be generalized to evaluate the security of protocol implementations other than SIP.

References

[1] CERT/CC Statistics 1988-2006, October 2006. http://www.cert.org/stats/cert_stats.html.

[2] Paul F. Roberts. Major Card Vendors Stay Mum on Data Breach, 2005. www.eweek.com.

[3] Mark Trumbull. AOL Security Breach Puts Web on Notice. *The Christian Science Monitor*, August 11 2006.

[4] The University of Texas at Austin Responds to Data Theft, April 2006. http://www.mccombs.utexas.edu/datatheft/.

[5] Rebecca Trounson. Major Breach of UCLA's Computer Files. *Los Angeles Times*, December 12 2006.

[6] A. Householder, K. Houle, and C. Dougherty. Computer Attack Trends Challenge Internet Security. *Internet Security (Supplement to Computer Magazine)*, 35(4):5–7, 2002.

[7] John Markhoff. Attack of the Zombie Computers is a Growing Threat, Experts Say. *New York Times*, January 7 2007.

[8] Brad Stone. Spam Doubles, Finding New Ways to Deliver Itself. *New York Times*, December 6 2006.

[9] Nicholas Ianelli and Aaron Hackworth. Botnets as a Vehicle for Online Crime. Technical report, CERT Coordination Center, 2005.

[10] Frank Piessens. A Taxonomy of Causes of Software Vulnerabilities in Internet Software. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 47–52, 2002.

[11] Sam Weber, Paul A. Karger, and Amit Paradkar. A Software Flaw Taxonomy: Aiming Tools at Security. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[12] OWASP. The Ten Most Critical Web Application Security Vulnerabilities. Technical report, 2004. The Open Web Application Security Project.

[13] U. Lindqvist and E. Jonsson. How to Systematically Classify Computer Security Intrusions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 154–163, Oakland, CA, USA, 1997.

[14] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys*, 26(3):211–254, 1994.

[15] Premkumar T. Devanbu and Stuart G. Stubblebine. Software Engineering for Security: a Roadmap. In *Proceedings of the International Conference on Software Engineering*, pages 227–239, 2000.

[16] Brian Krebs. Microsoft's Security Push Rolls on. *Washington Post*, October 6 2005.

[17] John Markhoff. Security Experts Say Risky Flaws Exist in New Microsoft System. *New York Times*, December 25 2006.

[18] P. Oehlert. Violating Assumptions with Fuzzing. *IEEE Security and Privacy Magazine*, 3(2):58–62, 2005.

[19] KPhone SIP Softphone. http://kphone.cvs.sourceforge.net/kphone/kphone/.

[20] H. Srinivasan and K. Sarac. A SIP Security Testing Framework. In *Proceedings of the IEEE Consumer Communications and Networking Conference*, pages 1–5, Las Vegas, Nevada, USA, 2009.

[21] Humberto Abdelnur, Olivier Festor, and Radu State. KiF: a Stateful SIP Fuzzer. In *Proceedings of the 1st ACM International Confer-*

ence on Principles, Systems and Applications of IP Telecommunications, pages 47–56, New York, USA, 2007. ACM Press.

[22] C. Wieser, M. Laakso, and H. Schulzrinne. Security Testing of SIP Implementations. Technical report, Columbia University, Department of Computer Science, 2003.

[23] 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 International Conference on Information Security*, volume 4176 of *Lecture Notes in Computer Science (LNCS)*, Samos Island, Greece, 2006. Springer.

[24] Voiper Security Toolkit. http://voiper.sourceforge.net/.

[25] Brian Chess and Gary McGraw. Static Analysis for Security. *IEEE Security and Privacy*, 2(6): 32–35, 2004.

[26] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proceedings of the 16th Annual Conference on Computer Security Applications*, pages 257–267, New Orleans, LA, USA, 2000.

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

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

[29] David Evans and David Larochelle. Improving Security using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002.

[30] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th Usenix Security Symposium*, Washington, DC, USA, 2001.

[31] Brian V. Chess. Improving Computer Security using Extended Static Checking. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 160–173, Berkeley, CA, USA, 2002.

[32] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 143–159, Berkeley, CA, USA, 2002.

[33] 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*, pages 201–220, Washington, DC, USA, 2001.

[34] J. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. *ACM SIGPLAN Notices*, 37(5):1–12, 2002.

[35] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, USA, 2000.

[36] H. Chen and D. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 235–244, Washington, DC, USA, 2002.

[37] 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 *Proceedings of the USENIX Security Conference*, pages 63–78, San Antonio, Texas, USA, 1998.

[38] T. Chiueh and F. Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the IEEE 21st International Conference on Distributed Computing Systems*, pages 409–417, Mesa, AZ, USA, 2001.

[39] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *Proceedings of the 10th USENIX Security Symposium*, pages 55–66, Washington, DC, USA, 2001.

[40] Crispin Cowan, Matt Baringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jaime Lokier. FormatGuard: Automatic Protection from printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 191–200, Washington, DC, USA, 2001.

[41] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel Protection from Temporary File Race Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 165–172, Washington, DC, USA, 2001.

[42] R. Jones and P. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the International Workshop on Automatic Debugging*, pages 13–26, 1997.

[43] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting Critical Elements of Stacks. White paper, 1999.

[44] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 251–262, San Diego, CA, USA, 2000.

[45] 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*, pages 1–13, San Jose, CA, USA, 1996.

[46] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 128–139, 2002.
[47] T. Jim, G. Morrisett, D. Grossman, and M. Hicks. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, USA, 2002.
[48] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, CA, USA, 1998.
[49] Ulfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 246–255, Berkeley, CA, USA, 2000.
[50] Jared DeMott. The Evolving Art of Fuzzing. In *DefCon*, 2006.
[51] B.P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of Unix Utilities. *Communications of the ACM*, 33(12), 1990.
[52] 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, 1995.
[53] J.E. Forrester and B.P. Miller. An Empirical Study of the Robustness of Windows NT Applications using Random Testing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 59–68, Seattle, Washington, USA, 2000.
[54] B.P. Miller, G. Cooksey, and F. Moore. An Empirical Study of the Robustness of MacOS Applications using Random Testing. *ACM SIGOPS Operating Systems Review*, 41(1):78–86, 2007.
[55] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proceedings of the 12th International Conference on World Wide Web*, pages 148–159, Budapest, Hungary, 2003.
[56] Leon Juranic. Using Fuzzing to Detect Security Vulnerabilities. Technical report, Infingo IS, 2006.
[57] Request for Comments 3261, Session Initiation Protocol. RFC Editor Database, http://www.rfc-editor.org/.
[58] X11::GUITest Libraries, Version 0.21. http://sourceforge.net/projects/x11guitest.

Ian G. Harris is currently an Associate Professor in the Computer Science Department at the University of California Irvine. He received his BS degree in Computer Science from Massachusetts Institute of Technology in 1990. He received his MS and PhD degrees in Computer Science from the University of California San Diego in 1992 and 1997 respectively. He was a member of the faculty in the Electrical and Computer Engineering Department at the University of Massachusetts Amherst from 1997 until June 2003. Professor Harris' research is related to testing of hardware and software systems. His field interest includes validation of hardware systems to ensure that the behavior of the system matches the intentions of the designer. He also investigates the application of testing for computer security. His group's security work includes testing software applications for security vulnerabilities and designing special-purpose hardware to detect intrusions on-line. Professor Harris serves on the program committees of several leading conferences in verification and security including IEEE/ACM Design Automation and Test in Europe (DATE), IEEE VLSI Test Symposium (VTS), IEEE Hardware Oriented Security and Trust (HOST), and the IEEE Workshop on High Level Design Validation and Test (HLDVT).

Thoulfekar Alrahem received his B.S. degree in Information and Computer Science from the University of California Irvine in 2007. He is currently working full-time as a software engineer in Southern California. Mr. Alrahem plans to pursue graduate school and join a Ph.D. program in the near future. His main interests fall in theory, algorithms, machine learning, and AI. When he is not coding, he likes to workout, run, read, and spend time with friends.

Alex Chen received his B.S. degree in Information and Computer Science from the University of California Irvine in 2007. He is currently working at Cedars Sinai Hospital under the Enterprise Information Systems department.

Nicholas DiGiuseppe has a BS in Information and Computer Science from the University of California, Irvine. He is striving to gain a PhD in the same field, and his interests include protocol fuzzing, testing, and the impact of games on technology and those playing them.

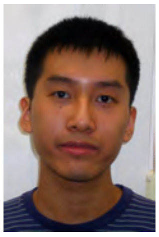
Jeffrey Gee graduated in 2008 from the University of California, Irvine, with a B.S. in Computer Science. Prior to graduation, he spent over a year researching security under Professor Ian Harris. In addition, he spent a year researching speech recognition at the Tokyo Institute of Technology, under Professor Sadaoki Furui, as a member of the Young Scientist Exchange Program.

Shang-Pin Hsiao received his B.S. degree in Computer Science from the University of California Irvine in 2007. His research interests include computer security, embedded systems, and grid computing.

Sean Mattox received his B.S. degree in Information and Computer Science from the University of California Irvine in 2007. His research interests include computer security, operating systems, and game development.

Taejoon Park received his B.S. degree in Information and Computer Engineering from the University of California Irvine in 2008. He is currently employed at Grandstream Inc. and is working on the testing of VOIP phones.

Saravanan Selvaraj received his B.S. degree in Computer Science and Engineering from the University of California Irvine in 2008.He performed an internship at Broadcom Inc. in Irvine, CA and is now employed in IT industry.



Albert Tam received his B.S. degree in Information and Computer Science from the University of California Irvine in 2007. His research interests include computer security, embedded systems, and web application development.

Marcel Carlsson is a security consultant working at FortConsult in Copenhagen, Denmark. He performs penetration testing and security auditing for International businesses and organizations. He has worked for both niche consultancies and international consulting firms performing security consulting in the US, UK and most European countries for more than 10 years. Marcel has an bachelor's degree in Electrical Engineering from the University of California San Diego, USA, and a Master's Degree from Chalmers, Gothenburg, Sweden. He is a validated PCI DSS and PA DSS auditor, has CISSP, CISA, CISM, GSNA certifications, enjoys attending hacking conferences, exploring and pwning technology in his spare time.