A SIP Security Testing Framework

Hemanth Srinivasan and Kamil Sarac
Department of Computer Science, University of Texas at Dallas
800 W Campbell Rd. Richardson, TX 75080, USA
{hbs051000, ksarac}@utdallas.edu

***Abstract*—Session Initiation Protocol (SIP) has emerged as the predominant protocol for setting up, maintaining, and terminating Voice over Internet Protocol (VoIP) sessions. In spite of the security mechanisms that it offers, several attacks are being made on the SIP architecture. In this paper we take a proactive approach and highlight the importance of testing SIP from a security perspective. We first give a brief introduction to some of the most common attacks on SIP. We then describe a framework to effectively test several security aspects of a SIP network and thereby help mitigate such attacks. We also present a genetic algorithm that we developed and used to generate data in our fuzz testing. Finally, we present the results of some tests performed on popular SIP devices using our framework.**

I. Introduction

With Voice over IP (VoIP), voice can now be transported on a traditional IP data network, making use of the abundance of the Internet and drastically reducing the cost of operation. However over the past few years VoIP services have been hampered by several security issues. With Internet being the carrier, VoIP networks are exposed to threats that an IP data network faces, e.g., eavesdropping, denial of service (DoS), IP spoofing [1], etc.

Session Initiation Protocol (SIP) has become the de-facto standard for VoIP services. It is described as *"an application-layer control protocol that can establish, modify, and terminate multimedia sessions (conferences) such as Internet telephony calls"* [2]. It is an ASCII/text based request-response protocol that works on a client-server model. The SIP standard [2] specifies several security mechanisms that can be used with native SIP. In spite of that, there have been several attacks on the SIP architecture with popular ones being DoS attacks [3] and malformed message attacks [4]. We believe that it is of utmost importance that each device in the architecture should be robust to potential attacks and this requires thorough testing of SIP implementations before deployment.

A comprehensive security testing should consider all known attack scenarios as much as possible. This requires a versatile security testing system that would allow the simulation of various attack scenarios during the testing phase. Our work is motivated by these factors and in this paper, we present a security testing framework that would allow one to test several security aspects of a SIP architecture. Our framework provides the ability to test SIP devices for vulnerabilities due to protocol non-conformance, performance limitations, DoS attacks, penetration/fuzz attacks and several others. The framework design considers flexibility and extensibility as a prime goal to provide an ability to customize test scenarios and incorporate new test techniques easily. By providing a common platform to integrate several test strategies, the framework also allows one to intelligently combine any of these techniques to generate new and more powerful test methodologies. As a part of the framework, we also present an algorithm to generate data to be used in effective fuzz testing. Finally we demonstrate the benefits of our framework by using it to test several VoIP clients for security vulnerabilities.

The rest of the paper is structured as follows. The next section is on the related work. Section III presents the common attacks on a SIP infrastructure. Section IV presents our proposed SIP security testing framework. Section V presents our Fuzz Data Generation Algorithm. Section VI presents our preliminary evaluations and Section VII concludes the paper.

II. Related Work

Several frameworks have been proposed to address specific aspects of SIP security. The SIP Forum Test Framework [5] is a conformance test suite that allows SIP device vendors to test their devices for common protocol errors and thus improve inter-operability. The SNOCER project [6], [4], [7] proposes a generic framework to protect SIP from malformed message attacks and describes a signature based detection mechanism for SIP message tampering. The PROTOS [8], [9] framework provides a fuzzing capability to assess the robustness of SIP implementations. The PROTOS SIP test suite tests for parser robustness mainly through syntax testing by crafting several malformed messages. Currently the suite has about 4500+ malformed SIP-INVITE messages. In [10], the authors present a stateful SIP/VoIP fuzz framework that incorporates protocol state transitions to make fuzzing more efficient. They also provide a grammar based approach to generate fuzz messages. In [11], several SIP testing tools are listed to test different components of a SIP infrastructure.

Most existing SIP security testing solutions target a specific aspect of SIP security or focus on a specific entity within a SIP architecture. The main difference between these solutions and the framework presented in this paper is that our framework aims at providing a comprehensive testing platform to integrate any aspect of SIP security testing into a uniform and easy to use platform.

III. Attacks on SIP Infrastructures

A SIP network architecture is typically composed of a SIP service infrastructure, an Internet gateway, a SIP aware firewall and a gateway to connect to other voice networks such as Public Switched Telephone Networks (PSTN). The SIP service infrastructure is composed of several entities including *SIP*

user agents (UA), a *registrar*, a *proxy server*, and a *redirect server*. An UA is the end point of a SIP connection and is either a SIP client or a SIP server. Every SIP client has to register with a *registrar* before it could participate in a SIP call. A *proxy server* is a call server that routes calls on behalf of a caller towards the target. Proxies are also useful for enforcing policy control (e.g., user authorization). A *redirect server* is also a call server that redirects the client to contact an alternate set of URIs. In the following, we summarize several important attack types on the SIP architecture and refer our reader to [12] for more details.

A. Message Flooding DoS Attacks

Message flooding DoS attacks are the most common attacks on the SIP architecture. Here an attacker tries to deplete resources on a server and hence deny service to legitimate users. REGISTER floods are aimed at the SIP registrar, INVITE floods target the SIP proxy/redirect server and authentication DoS affects either or both. DoS attacks are easy to launch requiring an attacker to simply craft a SIP message and send it. This can be achieved by using one of the available SIP test tools [11]. The popularity and relative ease of launching this type of attacks emphasize the need for proactive testing of SIP devices against them.

B. Message Flow DoS Attacks

Message flow based DoS attacks aim at causing a disruption to an ongoing call by impersonating one of the call participants. The SIP protocol defines a specific sequence of message exchanges for call setup and termination [2]. By sending a message out of its expected sequence, an attacker can disrupt the regular call flow.

C. Malformed Message Attacks

The text based nature of SIP makes it flexible in terms of extensibility of the protocol. On the other hand, it makes the implementations of the protocol vulnerable to malformed message attacks. SIP message parsers that process incoming messages have to be efficient to handle the degree of protocol flexibility and also be robust against malformed message attacks. Such attacks can be extremely dangerous as an incoming message may contain embedded shellcodes or malicious SQL statements that may cause significant damage (data loss or node compromise) at the victim site.

Finally, SIP is an application layer protocol and its implementations are vulnerable to common attacks to protocols at lower layers including IP and TCP/UDP. Other attacks on SIP infrastructure include attacks on DNS servers and SPam over Internet Telephony (SPIT) attacks.

IV. A SIP Security Testing Framework

Testing the security strength of a SIP architecture is a demanding process involving various test activities such as *Protocol Conformance Tests, Performance/Stress Tests, DoS Tests, Penetration/Fuzz Tests* and others. The process also requires continual upgrade to incorporate new techniques to test new attack scenarios. In addition to fulfilling the above requirements, a security testing framework must also be flexible, adaptable and conducive to the user's changing needs. We now present the architecture of our framework which aims to achieve these.

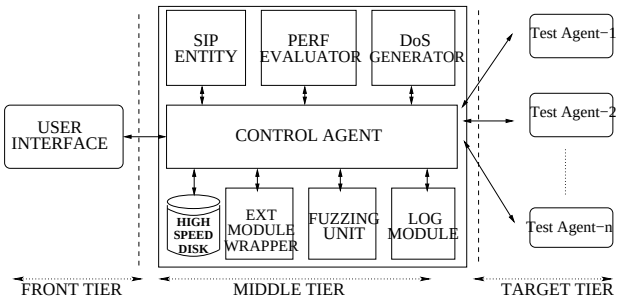

Fig. 1. SIP Security Test Framework

A. Framework Architecture

In this section, we present an outline of our SIP Security Testing Framework (SSTF). Our framework is presented in Figure 1 and is inspired by the work presented in [13]. The framework consists of three tiers as (1) a front tier that serves as the user interface, (2) a middle tier that constitutes the central control engine and other core components of the framework, and (3) a target tier that interacts with the test targets. Below are some details:

Front Tier
The front tier provides a uniform GUI to setup the test cases involving the different test modules incorporated into the middle tier. The GUI is dynamic in the sense that the user can fine tune it using configuration files. The control agent reads from the front tier and generates the test scenarios in an XML like format. Advanced users can specify finer details by directly manipulating the test scenario files.

Middle Tier
The middle tier forms the core of the framework. It consists of a central control agent and other modules that provide specific test functionalities as described below:

Control Agent. Control agent controls and orchestrates the entire test activity. It acts as an interface for each of the test modules and provides a uniform platform to integrate them. It also acts as an interface between the middle tier and the other tiers (front and target). The control agent loads the configuration files and instructs the front tier to generate an appropriate user interface. It also reads from the front tier and generates the XML like test scenarios that dictate a particular test flow. The control agent also instructs the test agents in the target tier to carry out the specific aspects of a test activity. For instance, if the test scenario involves testing multiple devices at the same time, the control agent would spawn multiple test agents and assign each test target to one of the test agents.

SIP Entity. SIP entity module is a full fledged SIP implementation conforming to SIP RFC [2] and capable of initiating sessions and responding to other SIP entities (e.g., test targets).

Control agent consults with the SIP entity when deciding on the result of a test case. Being the sanity controlling component, the SIP entity usually forms a part of any kind of test activity.

Performance Evaluator. This module provides various performance parameters to incorporate into a test case. Some common parameters include the number of calls in a test case, the call duration, and other dynamic rate control metrics. The control agent reads the supported metrics and presents them to the user through the front tier and thus dictates the test progress based on the user defined settings. The performance evaluator is a crucial component when estimating the performance benchmarks of a target device.

DoS Generator. The DoS generator provides support for simulating DoS attacks during a test session. It has the ability to spoof any given range of usernames and IP addresses. One effective test using the DoS generator is a Distributed DoS (DDoS) test where each call appears to come from a different source. One can also simulate a network wide DoS by varying the address of the test targets. The DoS generator is a key requirement for simulating DoS attacks and testing SIP-enabled firewalls and other security devices for intrusion detection.

Penetration Test/Fuzzing Unit. This unit is capable of altering a SIP message in malicious ways. The fuzzing unit provides the user with an in-built set of data to be used to construct a fuzzed message. It also allows to specify one's own fuzz data or auto-generate data using an underlying algorithm (see Section V). The control agent provides hooks to the fuzzing unit through configuration files where the user specifies the desired fields to be fuzzed and type of data to be used. Fuzzing unit also allows us to specify penetration test data instead of random fuzz data. The penetration test data could contain any vulnerability scanning information such as buffer overflow exploit shellcodes. Thus this unit plays a vital role in vulnerability scanning tests.

External Module Wrapper. This module serves as a wrapper for external tools. It interfaces the external tool and the control agent and allows for a mechanism to incorporate external tools into the framework. Several SIP test tools are available [11] that can be used to test various features of a SIP infrastructure. A wrapper module helps in building stronger test cases by making use of available stand alone SIP test tools.

Logging/Monitoring Module. This module monitors the test progress and logs key information. It assists the control agent to derive the test results and generate reports.

Target Tier

The test agents spawned by the control agent constitute the target tier. They are the direct interface to the test targets. Each test agent performs tasks based on the instructions from the control agent and then provides feedback. The distributed nature of the target tier has several advantages. A given test case can be distributed among different test agents thereby increasing the efficiency and effectiveness in terms of carrying out several tests in parallel. It also provides a level of redundancy; should a test agent fail/crash during a test, only that test is affected, the control agent and other test agents continue to function.

V. FUZZ DATA GENERATION

Fuzz testing or fuzzing is a software testing technique used to find implementation defects using malformed or semi-malformed input data. The outcome of a fuzz test, if successful, is usually a program crash possibly due to boundary check failure, buffer overflow, or some semantic error. For this reason, fuzzing is considered a valuable method in assessing the robustness and security vulnerabilities of software systems. The fuzz input data set can be very large; one may use *a brute force data set, a random data set, or a known problematic data set*. In most cases, we have to make a tradeoff between completeness and cost. One popular method is to use a combination of random data and a known list of problem causing data. Rather than using data directly from the above sets, we have come up with an algorithm to derive new fuzz data from these. We have categorized the data based on the SIP message format, into certain *fuzz data types* such as *SIP_int, SIP_ip, SIP_string,* etc. Each of these types can have data from any/all of above three data sets. To generate new data, we use a genetic algorithm [14] with a unique fitness function that ranks data based on their potential ability to expose defects. Higher the rank or fitness value of a data, higher is its probability of being used for new data generation.

Fuzz Data Generation Algorithm

- *Begin:* Choose initial (parent) population from one of our fuzz data types. We may choose each parent to be of the same type or of different types.
- *Fitness:* Evaluate the fitness of each data value in the population.
- *New Population:* Create a new population by repeating the following steps until the new population is complete
  - *Selection* Select one data value from each parent according to their fitness (better the fitness, better is the chance of being selected).
  - *Crossover:* Cross over the parents to form a new offspring. We use a single crossover point technique [14] with a random crossover point.
  - *Mutation:* Mutate the offspring at a chosen position, by altering a random number of bytes in each of the newly generated data values. This is to prevent the new population from being too similar and from completely deviating from its original type.
- *Acceptance:* Place the new offspring in a new population.
- *Improvisation:* Use the newly generated population for a further run of algorithm.
- *Test:* If the end condition is satisfied, stop and return the best solution in current population.
- *Repeat:* Go to Fitness computation step.

Fitness Computation

We know that fitness values determine what data we choose as

our initial population. We now define a set of parameters that contribute to the overall fitness value of a given data. All these parameters need not always be used; a subset of them can be used depending on the input population and the application being fuzzed.

- *Native Size*: Higher the deviation of the data from its original acceptable input size, higher is its contribution to the overall fitness value.
- *Native Type*: Higher the deviation from its original type (number, string etc), higher is the contribution.
- *Parents Fitness*: Either the maximum, minimum or average of the parents fitness values can be used.
- *Damage Co-efficient*: More damage a particular data has caused on previous occasions, higher is its damage co-efficient and contribution to the fitness.
- *Native Character Set*: Higher the presence of non-native (e.g., non-ASCII) characters, higher is the contribution.
- *User Defined*: User defined parameters depending on the application/protocol being tested. In many applications, format strings are very effective, so presence of such characters should increase the fitness level of that fuzz data. Similarly, Target/Attack specific data (e.g., SQL data in an SQL injection attack [7]) can also be taken into account.

Let N be the number of parameters chosen to contribute to the fitness value. The overall fitness value is calculated as follows:

1) Calculate the deviation factor DF = 1 / N. (We can also calculate a weighted DF, if some parameters need to be given more weight compared to others.)
2) Calculate the deviation contribution DC = A * DF, for each parameter, where A is the deviation percentage.
3) Calculate total deviation contribution TDC = SUM(DC) for all N.
4) Final Fitness Value F = Ceiling [TDC * 10].

VI. EVALUATIONS

In this section, we present the evaluations for a basic instantiation of our framework. Here we used SIPp [15] as our SIP entity module. The test topology consists of three entities including our framework, an open source stateful SIP proxy named Asterisk [16], and the device under the test (DUT). As an initial stage of testing and due to limited availability of resources in an academic setting, we use freely available evaluation copies of several popular SIP soft phone clients as the DUT. The softphones were installed on a PC running Microsoft Windows XP Service Pack 2 and Asterisk was run on an Intel P4, 2.6 GHz Processor, with 1GB of RAM. We plan to include other commercial SIP entities and security devices in our future tests. Since our testing activity is not complete, we have not informed the vendors about our test results. Hence, in this paper we do not disclose soft phone client names.

Performance Tests. Table I shows the results of some simple performance tests we conducted using our framework. Here we made calls to the UAs at different rates and then measured the response times and percentage of dropped calls if any. The response time was calculated by taking the average of the time taken for the first response message from the UA. We observed the following behavior: 1) some UAs (UA1) were not affected by the increased rates of calling; 2) at higher rates, the response times of some UAs (UA2) was very high, however the dropped call percentage was low; 3) in some UAs (UA3), increased rates caused both increased response times and dropped calls. These UAs usually *hanged* during the test which caused a sharp jump in response times.

Call Rate	UA1		UA2		UA3	
	Time(ms)	Drop(%)	T(ms)	D(%)	T(ms)	D(%)
1/s	1.9	0	0.9	0	1.8	0
5/s	2.4	0	0.8	0	2402	4
10/s	2.1	0	63.1	1	1481	33
50/s	2.1	0	59.9	3	1011	17

TABLE I
PERFORMANCE TESTS

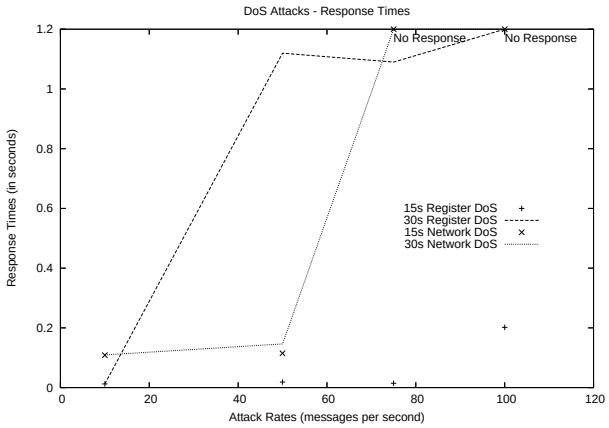

Fig. 2. DoS Attacks

DoS Tests. Figure 2 shows the results a DoS test performed on the Proxy. We considered two attack scenarios; 1) a *Register DoS* where REGISTER requests were made at different rates; 2) *a Network DoS* where bogus/genuine calls were made to the registered users at various rates. We measured the response time of the proxy for different attack durations. The response time for Register DoS was measured as the time taken to receive a 200 OK message for a single test request and the response time for Network DoS was measured as the time taken to receive a 180 RINGING message for a single genuine test call. From the results we observed that for Register DoS lasting 15s, the response times only marginally increased. However, for attacks lasting 30s or more, the response times increased more rapidly and at attack rates of 100cps or more, there was no response (i.e. message dropped). For Network DoS we found that the response times were much higher even for short duration attacks and calls were dropped at attack rates of 75cps and above. Thus, devices were less tolerant to Network DoS attacks.

Fuzz Data Generation Tests. Table II shows the results of the tests we carried out to validate our fuzz data generation

Parent 1	Parent 2	Generation	Failures Found
SIP_int	SIP_int	0	4
		1	8
		2	10
		3	8
SIP_string	SIP_string	0	4
		1	5
		2	4
		3	5
SIP_colon	SIP_slash	0 (colon)	1
		0 (slash)	1
		1	3
		2	3
		3	5

TABLE II
FUZZ GENERATION TESTS

algorithm. The test case consisted of making calls to the UAs at 1cps with fuzzed INVITE messages. We first carried out the tests using only our parent fuzz data set (Generation 0); we then carried out the same tests using offspring (Generation 1, 2, 3) data generated using our algorithm. The data used in these generations was not optimal; our intention was not to find a local/global optimal data set but to experiment our fitness function and generate new populations (data) and see if they were more, less or at least as effective as their parent populations. From the results we observed that the offsprings found many new failures that their parents did not find. This is a possible indication that our fitness function and parameters indeed ranked data based on their effectiveness. We also observed that the number of failures found in successive generations was never less than their original parent types. This indicates that our genetic algorithm indeed generated population that were an improved version of their parents.

Strezz Tests. Here, we consider mainly four types of tests: *1)* a simple stress test on the DUTs where we send 50 INVITE messages/sec; *2)* a simple fuzz test where we send a single INVITE message with one or more fields fuzzed; *3)* a call fuzz test where messages deeper in a call flow (e.g., ACK) were fuzzed; *4)* a Strezz (Stress+Fuzz) test where the test agent sends 5 fuzzed INVITE messages/sec.

Table III presents the results of the above tests. The first row shows that the tested DUTs handled 50 INVITE messages/sec and the second row shows that all the DUTs handled fuzzed INVITE messages without crashing. However, the third row shows that two of the tested DUTs failed when fuzzed messages were used beyond the initial INVITE. Finally, the last row shows that when fuzzing and performance testing were used together, most DUTs failed by crashing. These results are somehow surprising in that even though the DUTs pass the performance test with 50 INVITE messages/sec and the fuzz test with a simple fuzzed INVITE, most of them fail the Strezz test that combines these two testing scenarios. This sample test scenario thus demonstrates the importance of a comprehensive test environment that combines different test methods. We believe that many such effective test techniques can be developed using our framework to expose hidden vulnerabilities in a SIP service infrastructure.

Test	UA1	UA2	UA3	UA4	UA5
1	Pass	Pass	Pass	Pass	Pass
2	Pass	Pass	Pass	Pass	Pass
3	Pass	Pass	Pass	Fail/crash	Fail/freeze
4	Pass	Fail/crash	Fail/crash	Fail/crash	Fail/crash

TABLE III
STREZZ TESTS

VII. CONCLUSION

In this paper we have presented a framework for SIP security testing that provides a uniform platform to integrate several test methodologies and generate more effective test scenarios. Our intelligent fuzzer is not only protocol aware but also has an innovative algorithm to generate effective fuzz data and rank them based on their potential to expose defects. To the best of our knowledge, this is the first such framework for SIP security testing. Our evaluations have shown that devices that are resistant to individual stress and fuzz testing may still be vulnerable to test scenarios that combine these two test scenarios. These results demonstrate the importance of our claim that combining different test strategies produces much stronger test scenarios. Our evaluations of the fuzz data generation algorithm have shown that the ranking parameters and the fitness function that we designed depict the effectiveness of fuzz data to some extent.

REFERENCES

[1] S. Vuong and Y. Bai, "A survey of VoIP intrusions and intrusion detection systems," *6th Intl Conference on Advanced Communication Technology*, vol. 1, pp. 317–322, Sept 04.

[2] J. Rosenberg, H. Schulzrinne, and et al, "SIP: Session Initiation Protocol," Jul 02.

[3] D. Sisalem, J. Kuthan, and et al, "DoS Attacks Targeting a SIP VoIP Infrastructure: Attack Scenarios and Prevention Mechanisms," *IEEE Network*, vol. 20, no. 5, pp. 26–31, Oct 06.

[4] D. Geneiatakis, G. Kambourakis, and et al, "A Framework for Detecting Malformed Messages in SIP Networks," in *Proc of 14th IEEE Workshop on LANMAN*, Chania-Crete, Greece, Sept 05.

[5] "SIP Forum Test Framework (SFTF)," http://www.sipfoundry.org/sip-forum-test-framework/.

[6] "SNOCER: Low Cost Tools for Secure and Highly Available VoIP Communication Services," http://www.snocer.org/.

[7] D. Geneiatakis, D. Kambourakis, and et al, "SIP Message Tampering: THE SQL code INJECTION attack," in *Proc of 13th Intl Conf on Software, Telecoms and Comp Networks IEEE*, Split, Croatia, Sept 05.

[8] "PROTOS: Security Testing of Protocol Implementations," http://www.ee.oulu.fi/research/ouspg/protos/testing/c07/sip/.

[9] C. Wieser, M. Laakso, and et al, "Security testing of SIP implementations," Department of Computer Science, Columbia University, NY,, Tech. Rep., 03, http://compose.labri.fr/documentation/sip/Documentation/Papers/Security/Papers/462.pdf.

[10] J. Humberto, R. State, and et al, "KiF: A stateful SIP Fuzzer," in *Proc of IPTComm-Principles, Systems and Applications of IP Telecommunications*, New York, USA, Jul 07.

[11] "VOIP Security Alliance," http://www.voipsa.org/.

[12] D. Geneiatakis, T. Dagiouklas, and et al, "Survey of Security Vulnerabilities in SIP," *IEEE Communications Tutorials and Surveys*, vol. 8, no. 3, pp. 68–81, Oct 06.

[13] W.T.Tsai, L.Yu, and et al, "Scenario-Based Obj-Oriented Test Frmwrks for Testing Distributed Sys," in *11th IEEE Intl Workshop on Future Trends of Distributed Comp Sys*, San Juan, Puerto Rico, May 03.

[14] "Genetic Algorithms," http://en.wikipedia.org/wiki/Genetic_algorithm.

[15] "SIPp: Open Source test tool/traffic generator for the SIP protocol," http://sipp.sourceforge.net/.

[16] "Asterisk: Open Source PBX," http://www.asterisk.org/.