SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr

Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna

Department of Computer Science
University of California, Santa Barbara
{nomed, marco, rusvika, almeroth, kemm, vigna}@cs.ucsb.edu

Abstract. Fuzzing is a well-known black-box approach to the security testing of applications. Fuzzing has many advantages in terms of simplicity and effectiveness over more complex, expensive testing approaches. Unfortunately, current fuzzing tools suffer from a number of limitations, and, in particular, they provide little support for the fuzzing of stateful protocols.

In this paper, we present SNOOZE, a tool for building flexible, security-oriented, network protocol fuzzers. SNOOZE implements a stateful fuzzing approach that can be used to effectively identify security flaws in network protocol implementations. SNOOZE allows a tester to describe the stateful operation of a protocol and the messages that need to be generated in each state. In addition, SNOOZE provides attack-specific fuzzing primitives that allow a tester to focus on specific vulnerability classes. We used an initial prototype of the SNOOZE tool to test programs that implement the SIP protocol, with promising results. SNOOZE supported the creation of sophisticated fuzzing scenarios that were able to expose real-world bugs in the programs analyzed.

Keywords: Stateful Fuzzing, Network Protocols, Security Testing.

1 Introduction

Security is a critical factor in today's networked world. The complexity of many network protocols combined with time-to-deliver constraints imposed on developers and improper or insecure coding practices make errors inevitable. As a result, new vulnerabilities in network-based applications are found and advertised on a daily basis. The impact of vulnerability exploitation can be severe, and, in addition, the cost of correcting errors after a system has been deployed can be very high. Therefore, we need effective methods and tools to identify bugs in network-based applications before they are deployed on live networks.

One of the methodologies used to carry out this task is *fuzzing* [1,2,3]. Fuzzing is a form of black-box testing whose basic idea is to provide a system with unexpected, random, or faulty inputs, which expose corner cases not considered during implementation.

Fuzzing has a number of advantages over other testing techniques, such as manual code review, static analysis, and model checking. First, fuzzing can be

S.K. Katsikas et al. (Eds.): ISC 2006, LNCS 4176, pp. 343–358, 2006.
© Springer-Verlag Berlin Heidelberg 2006

applied to programs whose source code is not available. Second, fuzzing is largely independent of the internal complexity of the examined system, overcoming practical limits that prevent other testing methods (e.g., static analysis) from being able to operate on large applications. Being completely independent of the tested program's internals, the same fuzzing tool can be reused to test similar programs regardless of the language in which they are implemented. Finally, bugs found with fuzzing are reachable through user input, and, as a consequence, are exploitable.

A number of tools make use of fuzzing as a technique to test systems. They generally present limitations that hinder their wider and more effective use. In many cases, the means available to inject faults in the generated input are restrictive and do not include methods to specifically generate inputs that would likely trigger well-known, target-specific attacks. Furthermore, support for testing complex, stateful protocols is generally lacking; thus, requiring the tester to manually bring the system to the desired state before starting the actual test. Finally, the language adopted to describe how fuzzing should be performed is often very primitive, and, as a consequence, the activity of specifying fuzzing tests can require significant effort.

In this paper, we propose SNOOZE, a tool for building flexible, security-oriented, network protocol fuzzers. In SNOOZE we try to integrate the strengths of existing fuzzing tools, while correcting the limitations discussed above.

We have built a prototype of SNOOZE, and we used it to perform fuzzing of network applications that implement the Session Initiation Protocol (SIP) [4]. We decided to focus on SIP-based applications for several reasons. First, SIP is one of the core protocols of the VoIP infrastructure, which is becoming increasingly popular. Second, there are many competing implementations of SIP, some of which are not completely stable and have not undergone a full security assessment. Finally, SIP is a fairly complex, stateful protocol with many nuances and details that complicate its implementation and, therefore, its testing.

The contributions of this work are twofold:

1. We identify the requirements for a class of sophisticated fuzzers that can be used to test complex protocols.
2. We present the design and discuss the prototype implementation of a fuzzing approach that supports the testing of stateful protocols.

Our approach allows testers to build better fuzzers to evaluate more easily and more thoroughly the security strengths and weaknesses of complex, stateful protocol implementations. As a result, our tool can be used to limit the number and severity of vulnerabilities in deployed systems. We tested our tool on three real-world implementations of the SIP protocol, and we were able to identify previously unknown vulnerabilities.

The rest of the paper is organized as follows. In the next section we discuss the fundamental characteristics of fuzzing. In Section 3 we review related work. In Section 4 we present our approach and analyze the first prototype of SNOOZE. The evaluation of our tool is presented in Section 5. Section 6 concludes and discusses future work.

2 Background

Fuzzing is a black-box approach to testing the security properties of a software component. Fuzzing operates on the input and output of a component without requiring any knowledge of its internal working. The technique of fuzzing aims to expose flaws in applications by exercising them with invalid inputs.

Fuzzing requires three basic operations: generating random or unexpected input that could lead the application under test into an invalid state; injecting this input into the application; and, finally, observing whether the input causes the application to fail. Fuzzing relies on two fundamental assumptions:

1. A significant part of the faults contained in an application can be triggered through a limited number of input sources controlled by the user.
2. The execution of a faulty portion of an application manifests itself in visible ways, e.g., by producing unexpected output, crashing the application, or making it unresponsive.

This approach is different from white-box techniques, such as static analysis and model checking, where an explicit model of the tested application, or of some of its properties, is built and validated for correctness.

Fuzzing, unlike many white-box approaches, is not complete in the sense that it is not guaranteed to expose all faults in a program. On the other hand, all flaws found through fuzzing are guaranteed to correspond to some bug in the tested code, and, therefore, fuzzing can be considered as sound.

In general, there are two orthogonal strategies for creating faulty input for an application: generation and mutation. The generation strategy uses a formal specification of the input accepted by the tested system to generate a set of valid input values. These values are then modified by applying fuzzing primitives to obtain faulty test data. Mutation, on the other hand, relies on a set of valid input values (e.g., extracted from normal sessions), which, as before, are modified using fuzzing primitives. Generation requires that a formal specification of input values be available, but it is capable of generating all valid input. The efficacy of mutation, instead, is critically dependent on the completeness of the input set that is used. However, the generated input is generally more tractable and can focus on a specific area of weakness or type of flaw.

Fuzzers can also be differentiated on the basis of their level of understanding of input semantics. More sophisticated fuzzers automatically take into account rules constraining various parts of the input. For example, the value of some input parts may be dependent on characteristics of the whole input (e.g., checksums or content length fields), while other fields may be required to be encoded in particular formats (e.g., encrypted). Less sophisticated fuzzers leave the burden of taking care of these aspects to the user.

Fuzzers also differentiate themselves in the heuristics implemented to fuzz input and in their flexibility of use. Heuristics can be based on data types (e.g., for integer types, they may test boundary conditions such as large or small numbers, zero, and negative values) or on the expected vulnerability nature (e.g., SQL injection or format string). The complexity of applying fuzzing heuristics

can range from the invocation of a function call to the modification of an input grammar.

In some scenarios, faults in a system can only be reached after performing several intermediate steps, bringing the system to a certain state. For example, it might be necessary to perform a login step before gaining access to the application functionality that needs to be tested. Stateful fuzzers have knowledge of the system's state machine and are able to perform actions that differ depending on the current state. Stateless fuzzers, however, regard each input as completely independent. This is a substantial limitation and the main motivation behind the development of our stateful fuzzer.

3 Related Work

Fuzzing has been long used as a testing technique in areas not directly related to security (e.g., for reliability and fault tolerance assessment). One of the first uses of fuzzing is described by Miller et al. in [1]. In this paper the authors tested several standard UNIX utilities by giving them random input. The same methodology was used in later tests on the same applications [2] and on Windows [3] and MacOS [5] applications. All these tests make use of very simple fuzzing techniques, based on the generation of large chunks of random data, and have limited support for the testing of network protocol implementations.

Similar approaches proved useful when testing large, heterogeneous and complex systems, such as hardware components, real-time systems, and distributed applications. Loki, ORCHESTRA, and NFTAPE are significant examples of fault injectors[1] specifically designed for their environments. Loki allows one to inject faults in a distributed system on the basis of a partial view of the global state [6]. ORCHESTRA is a fault injection framework for distributed systems in which faults are specified as Tcl scripts, which are injected in a layered protocol stack [7]. NFTAPE adds support for multiple fault models and fault injection methods [8]. These approaches are very interesting but their focus is not on security.

More recently, fuzzing has been applied to the testing of web services. For example, one effort describes a fuzzer for web form-based services [9], while another presents dependability tests of SOAP components [10]. WSDigger is an open source tool for black-box testing of web services that takes the WSDL file of a web service as an input and tests the service with a specially crafted payload [11]. While the general ideas proposed in these works are probably applicable to different domains, they propose tools that are restricted to the testing of web services.

There are a number of tools that specifically target network protocols. The most representative of this class of fuzzers are SPIKE [12] and PROTOS [13]. The former, developed by Dave Aitel, is a framework which provides an API and set of tools to aid in the creation of network protocol fuzzers in C. In

[1] Fuzzing is usually considered to be a variant of the fault injection approach which uses randomized input.

SPIKE, a protocol packet is divided into a set of blocks, each of which can be fuzzed independently and automatically. Any change in a block size caused by a fuzzing transformation is handled automatically by SPIKE. However, the block abstraction provided by SPIKE is fairly low-level and does not allow one to easily model stateful protocols and complex messages, and their dependencies. In addition, because SPIKE-based fuzzers have to be implemented in C, their development can be effort-intensive and more complex than when using higher-level languages.

PROTOS, which was developed by the Oulu University Secure Programming Group, unlike SPIKE, does not provide an API for building custom fuzzers. Instead, it provides reusable test suites consisting of carefully crafted protocol-specific messages. Unlike other fuzzers that just send random input to a target system, PROTOS strives to generate input more intelligently by starting from the formal specification of a protocol and then using fuzzing values to generate faulty inputs. These inputs are based on heuristics that focus on triggering specific vulnerabilities, such as format string vulnerabilities and buffer overflows. The PROTOS approach has proven to be very effective: in 2002 it led to the discovery of many vulnerabilities in implementations of the Simple Network Management Protocol [14]. However, PROTOS does not provide fuzzing primitives or the ability to modify test cases without changing the protocol grammar itself, which can be a non-trivial task. Finally, it is difficult to completely evaluate because the engine used to generate test cases is not publicly available.

The goal of our project is to create a fuzzing tool that incorporates the best features of the existing fuzzers and, in addition, supports the creation of *stateful* protocol fuzzers. In the next section, we present the architecture of our fuzzing tool, which we call SNOOZE.

4 Architecture

SNOOZE is an extensible tool for the development of stateful network protocol fuzzers. It consists of the *Fault Injector*, the *Traffic Generator*, the *Protocol Specification Parser*, the *Interpreter*, the *State Machine Engine*, and the *Monitor*. Figure 1 shows the high-level architecture of SNOOZE.

The Interpreter is responsible for running the fuzzing tests. It takes as input a set of protocol specifications, a set of user-defined fuzzing scenarios, and a module implementing scenario primitives. A protocol specification defines the general characteristics of a protocol. These characteristics include, but are not limited to, the protocol type (e.g., whether it is binary or character based), the general format of header fields, the syntax of messages that can be exchanged in the protocol, and the allowed message flows (i.e., a state machine). The specification language is XML-based.

While SNOOZE has a number of protocol specifications included, new specifications can easily be added as needed. In addition, these specifications need only be written once, and they then can be reused to write testing scenarios. The Parser parses a protocol specification and makes it available to other parts of the tool.

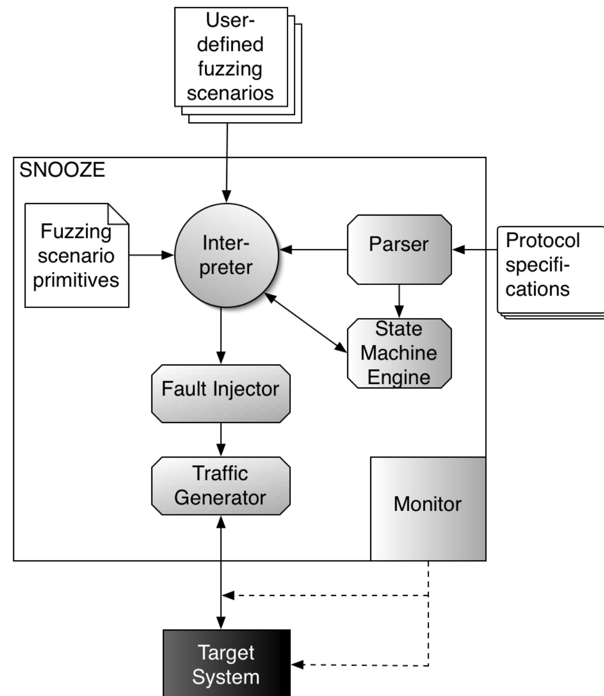

Fig. 1. Main components of SNOOZE

An example of a protocol specification is presented in Figure 2, which defines the syntax of the SIP INVITE message. In a protocol specification, each message is defined by a `<msg-rule>` element. Each `<msg-rule>` element consists of `<build-rule>` elements, which reference `<rule>` elements. The `<rule>` element either contains references to other building rules or specifies a default value for the corresponding message field. More specifically, in the example in Figure 2, the element `<build-rule id="SIP-Version"/>` specifies that each SIP INVITE message is required to contain a `SIP-Version` field, the syntax of which is defined by the `<rule>` element with ID `SIP-Version`. The default value for the `SIP-Version` field in this case is the string `SIP/2.0` concatenated with a value generated by the `CRLF` rule.

The default values assigned to fields are subject to change through the use of the mutation primitives described later. This makes it possible to modify the values of fields and to insert additional and user-defined fields into the message generated from the specification.

The dynamic aspects of a protocol, i.e., the valid sequences of exchanged messages, are specified with a state transition diagram. Each state represents a different step in the evolution of a conversation between two end-points: one state, for example, could record the fact that an INVITE message has been sent but the corresponding acknowledgment has not been received yet. The rules of

a protocol dictate the allowable transitions from one state to the other. For example, a rule would describe that when a CANCEL message is received the system should transition back to the initial state. Transitions are guarded by a condition that specifies which events can trigger the transition. Events can be the reception or the transmission of a message with specific values in determined message fields. Figure 3 shows a fragment of the specification of the SIP state diagram for a client user-agent.

```
<protocol type="ascii">
    <msg-rule id="INVITE">
        <build-rule id="INVITEm"/>
        <build-rule id="Request-URI"/>
        <build-rule id="SIP-Version"/>
        <build-rule id="Via"/>
        <build-rule id="Max-Forwards"/>
        <build-rule id="From"/>
        <build-rule id="To"/>
        <build-rule id="Call-ID"/>
        <build-rule id="CSeq"/>
        <build-rule id="Contact"/>
        <build-rule option="optional" max="inf" id="message-header"/>
        <build-rule id="Content-Length"/>
        <build-rule id="CRLF"/>
        <build-rule option="optional" max="1" id="message-body"/>
    </msg-rule>
    ...
    <rule id="SIP-Version">
        <field type="string">SIP/2.0</field>
        <build-rule id="CRLF"/>
    </rule>
    ...
</protocol>
```

Fig. 2. Part of the specification of the SIP INVITE message

In the current implementation of our tool, protocol specifications are manually extracted from standards, such as Request for Comments (RFC) documents, and can describe a protocol's features with the level of detail desired by the user. In addition, the specifications can be used by the user to define default values to be used for the various protocol fields.

Scenario primitives are the basic operations that are available for a user to test a system; that is, they are the building blocks to derive test "drivers". Currently, scenario primitives include mechanisms to build messages according to a protocol description, to send and wait for messages, to fuzz specific fields in a message, and to explore and leverage the state information available for a stateful protocol. Some of the available primitives are shown in Table 1.

The Fault Injector component allows a user to manipulate "normal" messages of a protocol in ways that, ideally, will cause faults in the target implementation.

```
<graph xmlns="http://www.martin-loetzsch.de/DOTML" id="SIP">
    <!-- states -->
    <node id="Start" root="true"/>
    <node id="Invite.Calling"/>
    <node id="Invite.Proceeding"/>
    <node id="Invite.Completed"/>
    <node id="Invite.CompletedAck"/>
    <node id="Terminated"/>
    ...

    <!-- transitions -->
    <edge from="Start" to="Invite.Calling">
        <send-message protocol="SIP" type="INVITE"/>
    </edge>
    <edge from="Invite.Calling" to="Invite.Proceeding">
        <recv-message protocol="SIP" type="RESPONSE">
            <field name="code" value="1??"/>
        </recv-message>
    </edge>

    ...
</graph>
```

Fig. 3. Part of the specification of the SIP state diagram

The current prototype includes a set of functions that can be used to fuzz string and integer fields in a scenario. The fuzzing functions implement various heuristics based on the testing of boundary conditions, such as very long strings, large numbers, or exploit inputs for common vulnerabilities such as SQL or command injection.

A fuzzing scenario encodes the fuzzing activity to be performed. A scenario uses the protocol specifications, scenario primitives, and the fuzzing module described above to build messages appropriate for a target protocol by fuzzing some of their fields and sending them to the target system. In the current implementation of our tool, a fuzzing scenario is a Python script that makes use of SNOOZE components and is run by the standard Python interpreter.

Figure 4 shows a complete, albeit simple, fuzzing scenario. In this scenario, we specify that we want to use SIP over UDP. We then build a SIP INVITE message with default values for every required field. After the INVITE message is built automatically by the SNOOZE engine, we set the `Request-URI` and `To` fields to some fixed value and specify that we want the `From` field to be fuzzed with values that are likely to expose an SQL injection vulnerability. The message is sent ten times using a loop. For each iteration of the loop, any piece of the `SnoozeMessage` that should be fuzzed, in this case, part of the `From` field, will contain a new fuzzed value that is automatically generated by the Fault Injector. Recall that the Fault Injector is responsible for performing fuzzing transformations on the data stored in a generic type (e.g., `SnoozeString`) as specified in that generic

Table 1. The SNOOZE primitives

Name	Description
snoozeUse	Parses the specification of the provided protocol
snoozeOpen	Opens a session with the given host and performs any required initialization
snoozeClose	Closes the given session and performs cleanup
SnoozeMessage	A class modeling protocol-independent messages
setField	Method of SnoozeMessage that allows one to set a field in a message to a given value
snoozeSend	Sends a message
snoozeExpect	Waits for a message
SnoozeString	Generic string type used in SnoozeMessage
SnoozeInt8	Generic eight bit integer type used in SnoozeMessage
SnoozeInt16	Generic sixteen bit integer type used in SnoozeMessage
SnoozeInt32	Generic thirty-two bit integer type used in SnoozeMessage
SnoozeInt64	Generic sixty-four bit integer type used in SnoozeMessage
fuzz_string_repeat	Fuzzes a field repeating a given pattern multiple times
fuzz_string_binary	Fuzzes a field inserting binary content
fuzz_string_x86nop	Fuzzes a field inserting x86 NOP instructions
fuzz_string_sql_inj	Fuzzes a field inserting strings likely to expose an SQL injection vulnerability
fuzz_string_sh_inj	Fuzzes a field inserting strings likely to expose a shell command injection vulnerability
fuzz_terminator	Fuzzes a field inserting a field terminator string
fuzz_intX_usig	Fuzzes a field inserting unsigned integer values. There exist versions for 8, 16, 32 and 64 bits integers
fuzz_intX_sig	Fuzzes a field inserting signed integer values. There exist versions for 8, 16, 32 and 64 bits integers
getValidSendMsgs	Returns the set of messages that may be validly sent in the current state of the protocol
getInvalidSendMsgs	Returns the set of messages that cannot be validly sent in the current state of the protocol
getValidReceiveMsgs	Returns the set of messages that may be validly received in the current state of the protocol
getInvalidReceiveMsgs	Returns the set of messages that cannot be validly received in the current state of the protocol
getCurrentState	Returns an object holding information about the current state of the protocol

type's constructor. At the end of the scenario, the session is closed. Figure 5 shows a selection of the messages generated by this scenario.

Figure 6 shows the use of some of the state-related primitives. As before, from the initial state, an INVITE message is sent. Since this represents a valid transition, the State Machine Engine updates the current state. The scenario, then, sends all messages that are not supposed to be sent from the current state and waits for a message. The scenario at this point could, for example, check the received packet to determine whether the invalid messages caused an unexpected transition in the implementation under test.

```
from snooze_scenario_primitives import *
from snooze_types import *

# fuzz SIP over UDP (the network profile)
profile = snoozeUse('SIP', 'udp')

host = '127.0.0.1'
port = 5060

sd = snoozeOpen(host, port, profile)

# build an INVITE message
m = SnoozeMessage('SIP', 'INVITE')
# modify default values of some fields
m.setField('Request-URI', [
    SnoozeString('ru', 'sip:test@' + host + ':' + str(port) + ' ')])
m.setField('To', [SnoozeString('tn', 'To: '),
    SnoozeString('tv', 'sip:test@' + host), SnoozeString('fe', '\r\n')])
m.setField('From', [SnoozeString('fn', 'From: '),
    SnoozeString('fr', 'sip:'),
    SnoozeString('ff', 'A', fuzz_string_sql_inj),
    SnoozeString('fv', '@' + host), SnoozeString('fe', '\r\n')])

for i in range(10):
    snoozeSend(sd, m)
snoozeClose(sd)
```

Fig. 4. An example fuzzing scenario

The `SnoozeExpect` primitive provides a mechanism to wait for messages, based on what type of protocol is being fuzzed (e.g., text or binary), the type of message, and the message's content. A scenario developer can then make conditional decisions based on the return value of the primitive, thereby navigating paths in the protocol state machine dynamically.

The operation of sending messages to the target system is performed by the Traffic Generator component. It receives messages created by a user scenario and transforms them into network packets while taking into account fields that need to be updated (e.g., checksums or content length fields). Then, it sends those packets to the target system.

The State Machine Engine keeps information about the state of network operations. In practice, it keeps track of transmitted and received messages, and it uses the protocol state diagram specification to check whether the messages trigger some transition from the current state to a new state.

Finally, the Monitor component analyzes the traffic data and the behavior of the target system, looking for manifestations of a fault. These manifestations include, but are not limited to, events such as a segmentation fault in the target system, a hang, an abnormal behavior, or an unexpected output that is the result of the system being put into an inconsistent state. In the

```
INVITE sip:test@127.0.0.1:5060 SIP/2.0
Via: SIP/2.0/TCP foo.cs.ucsb.edu:4040;branch=z9hG4bK74bf9
Max-Forwards: 70
From: sip:(SELECT%20*)@127.0.0.1
To: sip:test@127.0.0.1
Call-ID: UniQue1@tester.com
CSeq: 1 INVITE
Contact: <sip:whatever.com>
Content-Length: 0

INVITE sip:test@127.0.0.1:5060 SIP/2.0
Via: SIP/2.0/TCP foo.cs.ucsb.edu:4040;branch=z9hG4bK74bf9
Max-Forwards: 70
From: sip:%20OR%201=1@127.0.0.1
To: sip:test@127.0.0.1
Call-ID: UniQue1@tester.com
CSeq: 1 INVITE
Contact: <sip:whatever.com>
Content-Length: 0
```

Fig. 5. An example of the messages sent when executing the scenario in Figure 4

current SNOOZE prototype, rudimentary monitoring is available. This is provided through the `snoozeExpect` primitive, which will alert the tester when either an unexpected message is received or a timeout expires without receiving any data, which usually indicates the target system has crashed or hung because of the last `snoozeSend`. In addition to this automated monitoring, manual inspection must be done on the target system to identify whether the system is "behaving" correctly when a fault does not manifest itself in the messages being exchanged between the fuzzer and the target system.

5 Evaluation

Evaluating a fuzzer's performance is difficult. Generally, there is no direct feedback about the effectiveness of the fuzzer, other than the fact that the target system crashed or stopped functioning correctly. For this reason, the common evaluation practice is to run the fuzzer on a test suite of programs and evaluate its effectiveness based on the number of bugs found or the number of programs crashed. However, as discussed in previous sections, no conclusion can be derived about the completeness of the analysis performed through black-box testing. Therefore, an interesting extension to this basic practice would be to couple the number of bugs found with the amount of code exercised as a quantitative evaluation metric. We plan to investigate this extension in future tests as we believe that code coverage provides an estimate of how thorough the fuzzing process is. The assumption is that the more code paths that are traversed, the more potential bugs are discovered.

```
from snooze_scenario_primitives import *
from snooze_types import *

# fuzz SIP over UDP (the network profile). Enable the State Machine Engine
profile = snoozeUse('SIP', 'udp', 'client', True)
target_port = 5062
snooze_port = 5060

# open the session
sd = snoozeOpen('128.111.48.24', target_port, profile, snooze_port)

# build and send an INVITE message
m_inv = SnoozeMessage('SIP', 'INVITE', {'Content-Type': 'Content-Type'})
...[packet setup not shown]...
snoozeSend(sd, m_inv)

# send invalid messages
for msg in getInvalidSendMsgs():
    snoozeSend(sd, msg)

# wait for reply
snoozeExpect(sd)
...
```

Fig. 6. A scenario that uses state-related primitives

Qualitative metrics are also valuable. The ease of creating powerful fuzzing scenarios is a key factor in the adoption of one tool over another. In addition to providing fuzzing functionality, the ability to build simple, general-purpose clients is another metric to consider when comparing fuzzers.

Having decided on the appropriate metrics for evaluating our tool, we chose to focus our attention on the Session Initiation Protocol (SIP) [4]. SIP is an application-layer signaling protocol used to create, modify and terminate sessions with one or more participants, such as those found in Internet conferences and Internet telephone calls. Managing sessions involves multi-step operations. Consider for example the steps involved in the setup of a call: the caller sends an `INVITE` message to the callee; the user agent of the callee sends back a `Ringing` status response; when the user answers the call, an `OK` message is generated; the caller replies to this message with an `ACK` message. A similar exchange of messages is required for call termination. A consequence of the statefulness of SIP is that many bugs can be exposed only by exploring states that are "deep" in the protocol state machine, i.e., states that are reachable only after exchanging a coherent series of messages with the application under test.

We chose SIP for our evaluation for several reasons. First, there are several open-source implementations available. This allowed us to assemble a set of applications to test and to investigate the problems we found by code inspection. Second, SIP is not yet fully mature. Several implementations are still not

completely RFC-compliant and most projects have been started within the last couple of years. Finally, SIP is currently being used as the signaling protocol for many popular IP telephony, chat, and video conferencing applications.

We built a testbed of different SIP implementations consisting of the following programs: Linphone 1.1.0 [15] compiled with libosip 2.2.2 [16], Kphone 4.2 [17], and SJphone 2.99a [18]. These programs are a representative set of commonly used programs that utilize SIP.

Our tests consisted of running a scenario that fuzzed different combinations of fields, using all of the fuzzing primitives that are currently implemented in SNOOZE. The scenario explores different states of the programs under test, by sending the sequence of messages `INVITE, CANCEL, ACK`. Note that, in this way, we set up a complete SIP dialog, comprising several transitions in the SIP state machine, and we can perform fuzzing at all of the traversed states. We let this scenario replay this message sequence 19,000 times using different fuzzing values. This fuzzing scenario did not cause any fatal error, e.g., crash or hang, in SJphone or Kphone. However, it found several problems in Linphone and hereafter we describe three bugs that are representative of the types of flaws that SNOOZE can expose.

The first example is a crash caused by the initial `INVITE` message in our test sequence. Linphone shows the identity of a caller by presenting the content of the `From` field of a SIP `INVITE` message. When receiving fuzzed messages, the message parsing routine in Linphone is unable to parse the `From` field and returns a `NULL` value to its caller instead of a valid pointer to the parsed content. Unfortunately, the caller routine does not check the returned value and blindly dereferences it, causing a segfault and a subsequent crash of the program. Even though this error cannot generally be used to further escalate privileges, it can be considered a denial of service attack. The bug has been acknowledged by the author of the program and corrected in the following release [19].

A second crash resulted from a malformed `ACK` message, the last in the sequence that we were playing. The message, on its own, had no effect, and the bug only manifested itself in the case where there was an open call. That is, there was state saved in the form of a dialog. This bug, similar to the previous one, results from an attempted `NULL` pointer dereference in `libosip`. In this particular iteration of the scenario, an `INVITE` message had been sent which caused Linphone to enter the `ringing` state. The subsequent `CANCEL` message for that call was then ignored by Linphone because it was being fuzzed with binary data, making it non-parsable. A new call was then attempted with another `INVITE` message, causing Linphone to save the current state. Several message sequences later, an `ACK` message was sent with no `Call-ID` field present. Linphone received the parsed message from libosip, recognized that it was an `ACK`, and iterated through the dialogs associated with each phone call, calling the libosip routine `osip_dialog_match_as_uas`. The first step of this routine is to convert the `Call-ID` field of the received message to a string and then compare it to the same field in the stored dialog. In this case, the internal routine to convert the `Call-ID` field to a string returns -1, indicating an error, which

osip_dialog_match_as_uas fails to check. The resulting NULL pointer is then passed to strcmp which causes the segmentation fault to occur. This bug would not have been found without the ability to drive the application to a state deep in the SIP state machine.

A third crash was related specifically to the application's graphical interface. Although the details are not clear at this point, an improper series of messages cause debug statements of the form "Xlib: unexpected async reply" to be printed to the console. This problem is most likely caused by threading issues affecting the use of Xlib. The exact problem, however, is still to be determined.

From a qualitative point of view, SNOOZE, even in its first prototype version, has some advantages over other fuzzers. First, SNOOZE follows an object-oriented approach to the creation and manipulation of protocol messages by allowing the user to abstract away irrelevant details. This feature, coupled with the use of protocol specifications, greatly eases the task of dealing with messages. The result is that users can build valid protocol messages by simply invoking the SnoozeMessage constructor and message fields can later be manipulated by calling methods of the SnoozeMessage class. This contrasts with other fuzzers that require users to manually construct each message (e.g., SPIKE), and with those that allow users to modify messages only by manipulating the protocol grammar (e.g., PROTOS). Second, it provides a set of fuzzing methods that can be easily reused in multiple scenarios. Third, by design, it should be easy to use SNOOZE as the basis for general-purpose network clients, implemented using calls to SNOOZE primitives like SnoozeMessage, snoozeSend and snoozeExpect. This is not possible with fuzzing tools that only build test cases.

6 Conclusions

The complexity of current network protocols and the increasing number of attacks against protocol implementations require a stronger emphasis on the testing of programs. Not only more powerful but also more intuitive tools for assessing the security of network programs are needed.

We believe that fuzzing, i.e., injecting faults into a program by exercising it with random and unexpected input, can be a powerful testing tool. Even though fuzzing does not guarantee completeness of the analysis, it provides a practical way to quickly assess the robustness of an implementation to malicious input.

We described the initial design and implementation of SNOOZE, a tool for building flexible, security-oriented, multi-protocol network fuzzers. The first prototype of SNOOZE provides a set of primitives to flexibly generate fuzzed messages. It also provides support for stateful protocols, allowing for rapid development of fuzzing scenarios. The tool is protocol-independent and can be easily extended.

The preliminary results from using SNOOZE on a testbed of programs implementing the SIP protocol show that SNOOZE can be effectively used for finding bugs that are hidden deep in the implementation of stateful protocols. Moreover, the combination of reusable fuzzing primitives together with initial support for

stateful protocols allowed for implementation of quite complex stateful scenarios with reduced user effort.

In the future, we plan to extend SNOOZE in a number of directions. First, we plan to enhance the support for stateful protocols, particularly exploring ways to synchronize the state of the communication as seen by the fuzzer with the state of the application under test. Also, we plan to develop a GUI that will allow a scenario developer to build stateful scenarios graphically, in an intuitive manner. In addition, we intend to further evaluate SNOOZE using the code coverage metric. Finally, we will test the idea of composing SNOOZE with model checking tools to better model the exploration of the protocol state machine.

Acknowledgment

This research was supported by the Army Research Laboratory and the Army Research Office, under agreement DAAD19-01-1-0484. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

References

1. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the ACM **33**(12) (1990) 32–44
2. Miller, B.P., Koski, D., Lee, C., Maganty, V., Murthy, R., Natarajan, A., Steidl, J.: Fuzz Revisited: A Reexamination of the Reliability of UNIX Utilities and Services. Technical report, Computer Science Department, University of Wisconsin (1995)
3. Forrester, J.E., Miller, B.P.: An empirical study of the robustness of Windows NT applications using random testing. In: Proceedings of the 4th USENIX Windows Systems Symposium. (2000) 59–68
4. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: RFC 3261: SIP: Session Initiation Protocol (2002)
5. Miller, B.P., Cooksey, G., Moore, F.: An Empirical Study of the Robustness of MacOS Applications Using Random Testing. Technical report, Computer Science Department, University of Wisconsin (2006)
6. Cukier, M., Chandra, R., Henke, D., Pistole, J., Sanders, W.H.: Fault Injection Based on a Partial View of the Global State of a Distributed System. In: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, Washington, DC, USA, IEEE Computer Society (1999) 168–177
7. Dawson, S., Jahanian, F., Mitton, T.: ORCHESTRA: A fault injection environment for distributed systems. Technical Report CSE-TR-318-96, University of Michigan (1996)
8. Stott, D.T., Floering, B., Kalbarczyk, Z., Iyer, R.K.: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In: Proceedings of the 4th International Computer Performance and Dependability Symposium. (2000) 91–102
9. Huang, Y.W., Huang, S.K., Lin, T.P., Tsai, C.H.: Web Application Security Assessment by Fault Injection and Behavior Monitoring. In: Proceedings of the 12th International World Wide Web Conference, New York, NY, USA, ACM Press (2003) 148–159

10. Looker, N., Xu, J.: Assessing the Dependability of SOAP RPC-Based Web Services by Fault Injection. In: Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (2003)
11. Foundstone: WSDigger. (http://www.foundstone.com/resources/s3i_tools.htm)
12. Aitel, D.: The Advantages of Block-Based Protocol Analysis for Security Testing. Technical report, Immunity, Inc. (2003)
13. Kaksonen, R., Laakso, M., Takanen, A.: Software Security Assessment through Specification Mutations and Fault Injection. In: Proceedings of Communications and Multimedia Security Issues of the New Century. (2001)
14. Oulu University Secure Programming Group: PROTOS Test-Suite: c06-snmpv1. Technical report, University of Oulu, Electrical and Information Engineering (2002)
15. Linphone Project Team: Linphone: Telephony on Linux. (http://www.linphone.org/)
16. A. Moizard: The GNU oSIP library. (http://www.gnu.org/software/osip/)
17. KPhone Project Team: KPhone: a voice over internet phone. (http://sourceforge.net/projects/kphone/)
18. SJ Labs: SJphone. (http://www.sjlabs.com/sjp.html)
19. Morlat, S.: Re: [SNOOZE] remote crash of linphone-1.1.0. Personal communication to the authors (2006)