KiF: A stateful SIP Fuzzer

Humberto J. Abdelnur
LORIA - INRIA Lorraine
615, rue du jardin botanique
Villers-les-Nancy, France
Humberto.Abdelnur@loria.fr

Radu State
LORIA - INRIA Lorraine
615, rue du jardin botanique
Villers-les-Nancy, France
Radu.State@loria.fr

Olivier Festor
LORIA - INRIA Lorraine
615, rue du jardin botanique
Villers-les-Nancy, France
Olivier.Festor@loria.fr

ABSTRACT

With the recent evolution in the VoIP market, where more and more devices and services are being pushed on a very promising market, assuring their security becomes crucial. Among the most dangerous threats to VoIP, failures and bugs in the software implementation will still rank high on the list of vulnerabilities. In this paper we address the issue of detecting such vulnerabilities using a stateful fuzzer. We describe an automated attack approach capable to self-improve and to track the state context of a target device. We implemented our approach and were able to discover vulnerabilities in market leading and well known equipments and software.

Keywords

Software Testing Techniques, Protocol Fuzzer, VoIP Security, SIP Vulnerabilities

1. INTRODUCTION

Over the past few years, protocol fuzzing emerged as a key approach for discovering vulnerabilities in software implementations. The conceptual idea behind fuzzing is very simple: generate random and malicious input data and inject it in an application. This approach is different from the well established discipline of software testing [5] where functional verification is checked. In fuzzing, this functional testing is marginal; much more relevant is the goal to rapidly find potential vulnerabilities. Protocol fuzzing is important for two main reasons. Firstly, having an automated approach eases the overall analysis process. Such an process is usually tedious and time consuming, requiring advanced knowledge in software debugging and reverse engineering. Second, there are many cases where no access to the source code/binaries is possible, and where a "black box" type of testing is the only viable solution. The current generation of fuzzers is the first one, where most of the existing approaches rely on randomly injecting input data without taking into account the syntax, semantics and state of the targeted applications.

IPTCOMM '07, New York USA

ISBN: 978-1-60558-006-7

Such approaches are very useful when testing local applications within a confined environment (debugging session), but are hardly applicable when testing protocol/application stacks, like for instance a SIP stack. Although some results can be obtained (see Protos [14]), in many cases these are limited to the processing of one single message (INVITE) lacking the capability to track the state of the remote application and to use behavioral level information in the fuzzing process. Our work was motivated by these limits, and in this paper we describe a statefull and context aware fuzzer, its design, implementation and experimental results.

This paper is organized as follows. Section 2 describes the challenges involved in achieving a model to fuzz messages and to automatically evaluate their performance in the target entities. Section 3 describes the related work in the area of fuzzing, software testing and protocol state testing. Section 4 overviews the general protocol fuzzing framework. Section 5 shows the some key constructs of our fuzzing process: a fuzzer expression grammar and the associated evaluation strategy. Section 6 defines the automated construction of the representative state machine behavior of the SIP protocol, the techniques required for a passive testing of the conformance of the target entity with the learned behavior and the simulation process of one end-point entities in a session. Section 7 details the implementation of the assessment platform and the results obtained in testing different SIP devices. Finally, section 8 concludes the paper and highlights future works for our research.

2. CHALLENGES

Fuzzing is an important topic in the context of security assessment, software testing and black box testing approaches. The major idea behind fuzzing is that input data will be tampered with random payload and will be injected in order to test the data validation and processing of a target application. Several ways exist in order to generate the injected data. The data can be generated from the scratch, by mutating existing valid data item or obtained by merging existing data. These possibilities do not represent the challenge itself, because random functions can always generate such data. The main challenge however resides in how to create input data that can reveal software errors and/or noncompliance to standards.

From the point of view of a tested device, two main logical structures can be considered (as illustrated in figure 1)

1. The unit which parses and translates a message to a structured format

2. the unit that will process that structured data and define the behavior to be executed

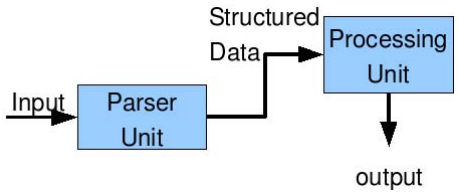

Figure 1: General Device Functionality

Assuming the generic structures of Figure 1 a crafted message can be classified based on the effect on the target:

1. Messages that are not syntactically compliant and are therefore rejected by the parser in the target entity.
2. Messages that are not syntactically compliant, where the parser does not detect such irregularities, but however the processing unit reject them.
3. Messages syntactically not compliant, where neither the parser nor the processing unit are able detect the irregularities.
4. Syntactically compliant messages containing semantic irregularities which are rejected by the parser.
5. Syntactically compliant messages containing semantic irregularities that are rejected by the the processing unit.

The first and fifth types are directly considered as garbage because they do not have any effect on the tested entity. The second type instead, allows the message to be processed but its failure is detected in an upper layer. The message itself did not provoke a mis-functionality in the target device but reveals a potential security hole. New messages may dig further to find more serious problems in the corresponding unit. In case of the third type, two consequences might arise: either a vulnerability is found or the information crafted in the message is not of concern for the entity. The latter may be the case of a proxy, which usually ignores the fields that are of no interest to speed up the process. The fourth type is associated with messages that may restrict the interoperability with other devices.

Our first objective was to define a flexible technique capable to generate messages of any of these types. We wanted to do more that current fuzzers, which in most cases are restricted to simple text based substitutions of large data chunks and/or injected format string attacks required to test for common buffer and format string vulnerabilities.

Some of the existing fuzzers use simplistic operational models, while others provide a rather complex interface, requiring major work to adapt them for additional tests. This last issue was one driving force in our work. We decided to research how complex fuzzers can be build on top of a small set of evolving and adaptive key building blocks.

Our aim was to provide a self-learning fuzzer that can evolve and use structural domain specific knowledge. Evolution is a key design feature required to build smart protocol fuzzers, while a domain (SIP) specific approach is more probable to provide better results.

The second challenge that we addressed was how to evaluate the effectiveness of generated fuzzed input. For this issue, some ideas can be found in the research papers on fuzzers and software testing [5, 14, 12, 10, 13]. The major issue is how to automatically detect that a fuzzed message was successful. If a device crashes, then probably checking its status before the reboot, might detect the crash. Checking the online status for embedded devices is not that trivial. For more complex network appliances, ICMP and/or application level port scanning are useful, but in case of VoIP devices where typically few ports are open and the implementation of the TCP/IP stack is fragile, few reliable approaches exist.

Our final interest was set by the idea to be able to fuzz at a protocol behavior level rather than only syntactically.

3. RELATED WORKS

Among the pioneering work in the field, the Mini-Simulation Toolkit by R. Kaksonen in [9] proposed an excellent framework for automatically generating crafted messages with a certain knowledge of states. It assumes in certain knowledge of the protocol to be tested, and an additional grammar which merges the syntax message and the transition behavior. The same work defines some semantic rules which allow to produce calculated fields like for instance the checksum. However, the syntax grammar and the state protocol are mixed in a same definition, which in fact provides a reduced set of scenarios. Another issue is that the rules to create exceptional messages are limited in functionality. Another successful approach is the one followed by D. Aitel in SPIKE [2, 3], where the concept of block-based fuzzing is introduced. This is based on the fact that protocols are always composed of the same primitives: invariants, blocks and variants. In this case the invariants are kept intact and the block are filled with fuzzed data. However, SPIKE is too low level, so it becomes highly effort-consuming when applied to complex protocols. An academic research by G. Banks et al. called SNOOZE [4] claims to be a stateful protocol fuzzer. It is based in user defined scenarios and a protocol specification. However, the protocol specification as well as the use cases are highly complex to describe while the operations to fuzz the data are limiting. Meanwhile, the stateful concept of the approach is not clear. By contrast, the approach presented by S. Emblenton in Sidewinder [7], describes a potential approach for the new generation of fuzzers, where a grammar specifying the syntax is provided and rules to evolve it according to the obtained results are presented. Meanwhile, a list of most popular fuzzers can be found on the ThreatMind[1] website.

Very few fuzzer approaches consider the evaluation of the effect of the crafted message. For instance, the work led by the Mini-Simulation Toolkit as the Protos SIP test-suite [14] stands out for its deployment and large test cases. Protos uses a set of test cases for which the analysis of success was done using in/out of band monitoring instrumentation. The research by P.M. Maurer at [12] also proposed several ideas of different approaches to evaluate the impact of the crafted messages. The first consists in generating the crafted message and anticipating what the regular reply should look like. This approach requires knowledge of the protocol and the effect that the crafted data may have. The second ap-

[1]http://www.scadasec.net/secwiki/FuzzingTools

proach proposed is to test more than one entity at the same time, where all of them should receive the same message and each one will reply according to its stack. If the replies are different, there is a chance that one of the entities is not respecting the protocol definition or abusing the freedom that it leaves, and therefore, vulnerabilities may be found. David Lee reveals in [10] a technique to test the data portions of a protocol. That work uses a state machine describing the state in which the protocol is at the current moment. Based on the types and properties of the incoming/outgoing messages, transitions of states are done. Each transition that is not compliant with the ones described by the state machine are considered as a protocol error in the implementation. An approach similar to the previous by H. Sengar [13] used for VoIP Intrusion Detection describes also the use of state machine to find inconsistencies in the message transition to detect possible attackers. Our previous article [1] proposes a generic and open framework for VoIP assessment operations able to extract information from a VoIP network and launch full fledged penetration tests.

4. FUZZING FRAMEWORK

The test validation process described in this article is illustrated in Figure 2. It consists in two autonomous components, the Syntax Fuzzer and the State Protocol Evaluator, which jointly provide a stateful data validation entity. A test is generated by a scenario, where a scenario represents a high level goal. For instance, a scenario can be to test SIP verbs (*INVITE, REGISTER*, etc. transactions). A scenario represents a series of tests. Scenarios are based on domain knowledge specific to the protocol and random data injection.

The tests may be similar to the normal behavior or can flood the device with malicious input data. Such malicious data can be syntactically non compliant (with respect to the protocol data units), or contain semantic and content wide attack payload (buffer overflows, integer overflows, formatted strings, or heap overflows).

The domain knowledge of SIP consists in a Context-Free grammar ABNF [6] used to describe the exact syntax of messages. An additional state machine (Protocol State Machine) is used to model the transitions in the system. These transitions are executed based on the incoming/outgoing messages. The syntax fuzzer takes a Fuzzer Evaluator and the provided syntax grammar to generate new and crafted messages. The Fuzzer Evaluator may depend on the State Protocol Evaluator in order to generate the final message (appropriated or not) to be sent to the target entity.

The State Protocol Evaluator requires a second state machine, called Testing State Machine. The latter provides the scenario, where some transitions should be chosen with more priority than others. The behavior or time-out events are also described. In cases where the actual transition is not represented in this second state machine, if it is allowed, the underlying protocol state machine can take control to properly finish the tests. Each transition modifies in fact the overall environment state of the system.

Figure 2 also shows the functional framework of the approach, where in the first example a SIP phone initiates a session by sending an INVITE. Our User Agent Server (UAS) processes the message and informs the State Protocol Evaluator. The latter induces the message that should follow. This message is constructed by the Fuzzer Evaluator according to the defined rules. Note that if the State Protocol Evaluator decides that another message should be received in order to proceed, the Fuzzer Evaluator will remain idle.

The traditional approach in the fuzzing community is by data input validation. This is done by generating crafted messages and observing the resulting behavior in the target entity. Generally, the resulting behavior is observed in terms of "aliveness factors", ie. state of the device: crashed or functional. With the help of the Protocol State Evaluator we can extend the analysis by observing incorrect transition over the states and observe responses which are not syntactically compliant.

4.1 Stateful Fuzzing Evaluation

Three techniques to evaluate the effectiveness of crafted messages in a target entity have been investigated in our work:

1. The normal behavior of the target entity should be learned for testing its aliveness in case it crashes during the tests. This alive behavior may be obtained by sending an *OPTIONS* message and observing its replies, if any. Some entities may not support or be configured to ignore such messages. For these cases another sequence of messages may be send as *REGISTER* or *INVITE and CANCEL* to allow to learn the normal behavior. This sequence is sent several times before starting the testing in order to assure that the entity replies always in the same way. It is important to note that such messages are not crafted because their only purpose is to evaluate the aliveness and correct functionality of the target entity.

2. The testing of the target functionality consists in a defined state machine with sequences of messages - see the Testing State Machine in figure 2. This state machine represents the scenario describing how the evaluator should react to specific events. It may describe the behavior after unexpected messages, timeouts or normal events. In the case where some transitions are not defined in scenario state machine, the underlying protocol state machine can take control in order to properly finish the transaction.

3. Finally, when the test is finished, it is also necessary to check if the device is alive as well as if it behaving in an usual manner. Note that a test may finish by timeout which does not really mean that the device crashed, but that the crafted message was too incorrect to be replied to. For this latter case, every time a test is launched, the alive tester may try to detect that the target entity is either alive or that it is still coherent with its initial learned behavior. Once this step is concluded, errors are either reported or it continues with step 2.

4.2 Reporting Events

Events are reported in one of the following cases:

- If a message generated by the target entity is capable to generate a transition that is not recognized by the Protocol State Machine, i.e. the last or previous messages provokes in the target entity a state where the protocol specification is violated.

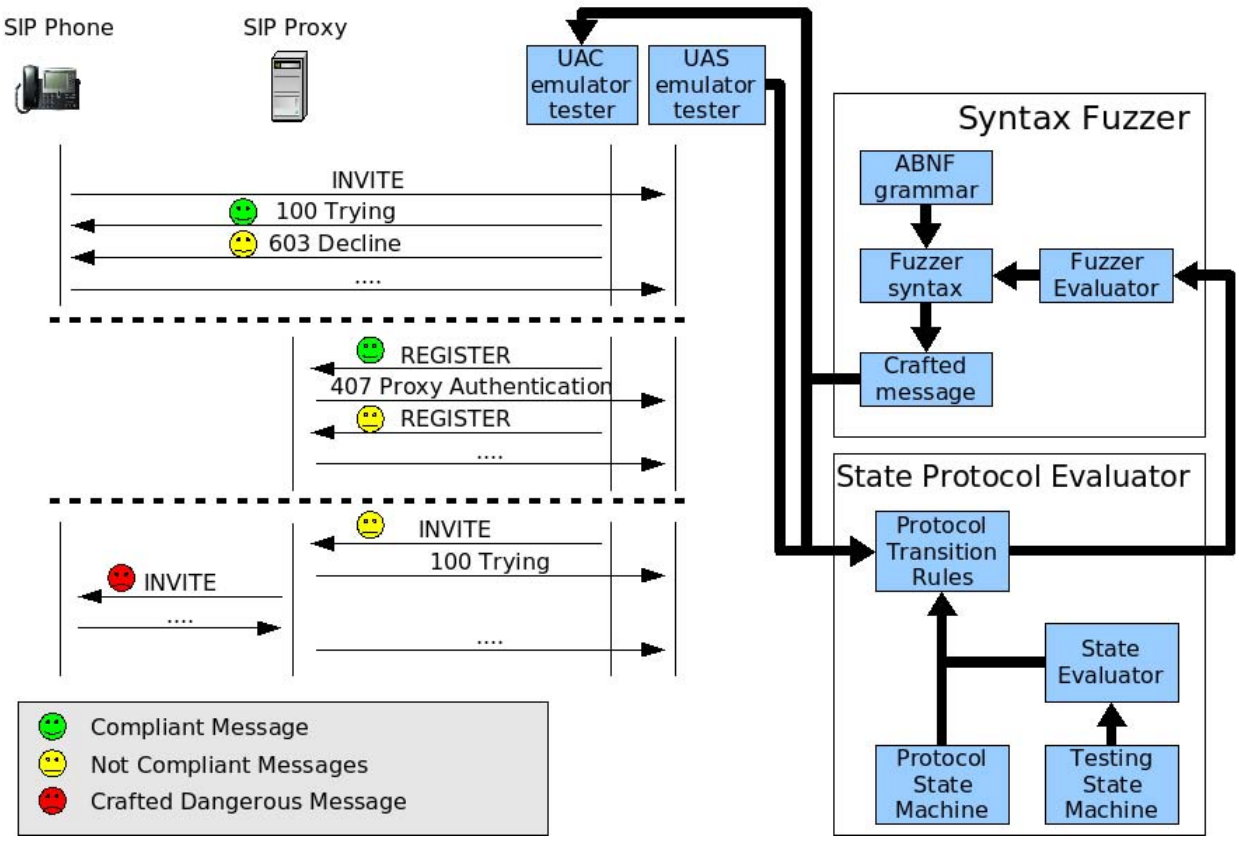

Figure 2: Fuzzing Framework

- If a message generated by the target entity is not compliant with the protocol syntax, i.e. the information was not well interpreted or just it was not considered at all, as it is the cases of some proxies. This is considered a good starting point to dig for vulnerabilities.
- Finally, when the aliveness tests are not responding as they should, either because no answer at all is obtained from the target entity or if a different one with respect to the already learned one is got.

5. FUZZER EXPRESSION GRAMMAR

Fuzzer are often classified based on multiple criteria: their speed to generate messages, the capability to discover known and/or new vulnerabilities, the quantity of tests that can be generated or even by the complexity of substitutions that they can perform taking into account a description of the protocol. We consider in this paper a more formal approach where a Fuzzer Expression Grammar is defined in order to describe the coverage of randomness in the generated message. This definition is closely related to the Parsing Expression Grammars [8], which formalizes the parsing grammar concepts.

Another important fact of fuzzers is related to the inputs that have to be provided in order to launch the test. Such inputs will define the generality, the specificity and the overall behavior. A certain type of grammar is required as well as knowledge about the syntax of the messages and the possible variable fields that may be changed by the fuzzer. Most of the time they require a lot of information and cover only a small scenario of generated data. It is also hard to know if the compliance with the protocol is kept or not. Very often, the rules to randomize such fuzziness may be either to simplistic and limited or to complex to be used in the creation of new tests.

Our fuzzing approach takes two inputs. The first is a ABNF (Augmented BackusâĂŞNaur Form) grammar [6], which is the standard syntax definition of a protocol specification. Thus, the fuzzer provides the flexibility to be adapted to different protocols. It's capable to generate messages compliant or not with the underlying grammar based on the second input: the Fuzzer Evaluator Interface.

The main work described in this paper consists in an in-depth application of our approach for the SIP protocol in order to check the implementation of SIP stacks within hardphones and SIP proxies.

5.1 ABNF Grammars

An ABNF is a grammar mostly used to formally describe the syntax of a protocol.

Formally a grammar of the type consists of 4-tuple $G = (\Sigma, N, P, n_0)$ where:

Σ = finite set of terminals (string literals).
N = finite set of non-Terminals.
P = finite set of mapping rules of the form $P : N \rightarrow e$, where e is an *expression* as described bellow.
n_0 is a non-Terminal called the starting symbol.

An inductive definition of the *expressions*, where it is as-

sumed that $e,\ e_1,\ ..,\ e_n$ are *expressions* as well, is as follows:

- Terminals.
- non-Terminals.
- Sequences $e_1\ ..\ e_n$
- Choices $e_1\ /\ ..\ /\ e_n$
- k-Repetitions $e^{(i,j)}$ where $0 \leq i \leq k \leq j$

Note that some assumptions were made for the sake of simplicity. All the other expressions not mentioned (Character Classes Terminals, Incremental Choices, Sequence Groups, Specific Repetitions and Optional Sequences) can be simulated by the previous expressions.

5.2 Fuzzer Expression Grammars

A Fuzzer Expression Grammar inherits from an ABNF grammar, being inherently linked to the underlying grammar. An additional evaluator exists in a Fuzzer Expression Grammar. This evaluator is guiding the reduction of the rules in order to generate a new message. As it will be explained later, this process may decide whether to be compliant or not with the syntax of the grammar.

A Fuzzer Expression Grammar consists of a 5-tuple $G = (\Sigma, N, P, E, n_0)$ where all the components $\Sigma,\ N$ and P are the same of the ABNF grammar and the evaluator E is:

E = Fuzzer evaluator of the form $E : e \times \theta \rightarrow \Sigma^*$ where e is a *fuzzing* and θ is the environment state.

A message m generated by this fuzzing grammar is

$$m = E(n_0)$$

thus, the typical objective of such a message is to represent a data input validation test for a protocol implementation instance.

5.3 Expressive Power

In order to formalize the expressiveness of the approach, an evaluation interface is defined in six main functions:

- $\mathcal{T} : \Sigma \times \theta \rightarrow e \times \theta$, which may replace a Terminal by another items.
- $\mathcal{N} : N \times \theta \rightarrow e \times \theta$, which may replace a Non-Terminal by another items.
- $\mathcal{C} : e_1\ /\ ..\ /\ e_n \times \theta \rightarrow \mathbb{N}_{\{1,n\}}$, which decides which item index should be chosen.
- $\mathcal{R} : e^{(i,j)} \times \theta \rightarrow \mathbb{N}_{\{i,j\}}$, which decides how many repetitions should be reduced.
- $\mathcal{S} : e \times \mathbb{N} \times \theta \rightarrow e \times \theta$, which may replace the i-item of the Sequence by another items.
- $\mathcal{I} : e \times \mathbb{N} \times \theta \rightarrow e \times \theta$, which may replace the i-repetition of the Repetition by another items.

Note that if there is a sense of location of the current reduction represented in the state, the latest two functions are just simple syntax extension of the combination of the four firsts. All these components interact with the evaluator in order to generate a new fuzzed message:

$$E, E_1 : e \times \theta \rightarrow \Sigma^*$$

$$E_1(e,\sigma) = \begin{cases} e & \text{if } e \in \Sigma^* \\ E(e,\sigma) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
&E(e,\sigma) = E_1 \circ \mathcal{T}(e,\sigma) && \text{if } e \in \Sigma \cup \{\varepsilon\} \\
&E(e,\sigma) = E_1 \circ \mathcal{N}(e,\sigma) && \text{if } e \in N \\
&E(e_1\ ..\ e_n,\sigma) = E_1 \circ \mathcal{S}(e_1,1,\sigma) \\
&\qquad\qquad ..\ E_1 \circ \mathcal{S}(e_n,n,\sigma) \\
&E(e_1/../e_n,\sigma) = E(e_i,\sigma) && \text{where } 1 \leq i \leq n \text{ and} \\
&\qquad && \ i = \mathcal{C}(e_1\ /\ ..\ /\ e_n,\sigma) \\
&E(e^{(i,j)},\sigma) = E_1 \circ \mathcal{I}(e,1,\sigma) \\
&\qquad\qquad ..\ E_1 \circ \mathcal{I}(e,k,\sigma) && \text{where } k = \mathcal{R}(e^{(i,j)},\sigma)
\end{aligned}$$

For the functions $\mathcal{T}, \mathcal{N}, \mathcal{S}, \mathcal{I}$ of the evaluator, five operations were defined that can help to progressively construct the Σ^* based in the Fuzzer Evaluation itself. These operations are described below:

- Produce either a fixed string or a random one generated from a regular expression.
- Append *expression* productions generated by another evaluator.
- Generate any rule defined by the grammar, which may be not the ones allowed in the current reduction.
- Generate a new rule defined on the fly, allowing the evolution of rules or addition of new ones.
- Generate a function rule. A function is an special case, because it escapes from the syntax concepts to define semantic actions. It is evaluated after the whole message has been reduced to Σ^* union other functions. Based on other items generated for the message, it generates the Σ^* appropriated for its current field. This function can be useful to add fields like checksum, content lengths, etc. However, infinite recursion has to be prevented.

The reduction of the expressions proceeds in a Depth First Search (DFS), where the generated message may be viewed as a tree (Figure 3), such that all the internal nodes are non-Terminal (e_5), Choices (e_2), Sequences (e_1 and e_4) or Repetitions (e_7) items and the leafs are Terminals (e_3, e_6 and e_8) or functions before being evaluated (e_9, where the functions set is denoted as F in the figure). In this way, each reduction branch can be uniquely identified in the tree by the path from the root to the current position. Definition 1 formalizes the *reduction path* concept.

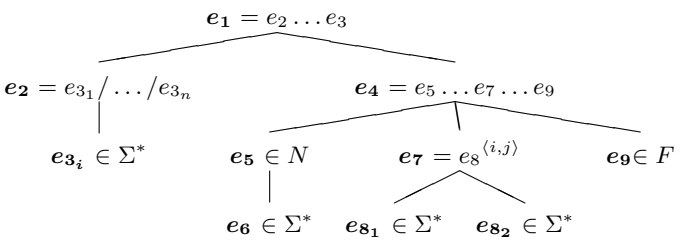

Figure 3: Tree reduction

Definition 1. A *reduction path*, $x \triangleright xs$, will define the steps for which an expression reduces to another (i.e. from an expression e_i to arrive to the expression e_j). Each step is defined by the relation $\Rightarrow_F$ as:

$$
\begin{array}{ll}
(e, x \triangleright xs) \Rightarrow_F (\mathcal{T}(e), xs) & \text{if } e \in \Sigma \cup \{\varepsilon\} \text{ and } \mathcal{T}(e) = x \\
(e, x \triangleright xs) \Rightarrow_F (\mathcal{N}(e), xs) & \text{if } e \in N \text{ and } \mathcal{N}(e) = x \\
(e_1 \,..\, e_n, x \triangleright xs) \Rightarrow_F (\mathcal{S}(e_x), xs) & \text{if } 1 \leq x \leq n \\
(e_1/../e_n, x \triangleright xs) \Rightarrow_F (e_x, xs) & \text{if } 1 \leq x \leq n \\
(e^{(i,j)}, x \triangleright xs) \Rightarrow_F (\mathcal{I}(e, x), xs) & \text{if } i \leq x \leq j
\end{array}
$$

and it is said that the *reduction path* $x \triangleright xs$ success from e_i to e_j if the $\Rightarrow_F$ closure is equal to

$$(e_i, x \triangleright xs) \Rightarrow_F^* (e_j, [])$$

5.4 Example Evaluators

In this section, we will look at two example evaluators for generating messages compliant with the underlying grammar (see section 5.4.1), and to generate messages based on the merging of different other messages (see section 5.4.2).

5.4.1 Compliant Grammar Evaluator

A definition of the inner functions of E, which randomly creates well formatted messages according to the specified grammar may be like follows (to simplify the example no environment state is used).

$$E : e \rightarrow \Sigma^*$$

$$
\begin{array}{ll}
\mathcal{T}(e) = e & \\
\mathcal{N}(e) = P(e) & \\
\mathcal{C}(e_1/../e_n) = i & \text{where } 1 \leq i \leq n \text{ chosen randomly} \\
\mathcal{R}(e^{(i,j)}) = k & \text{where } 1 \leq i \leq k \leq j \text{ chosen randomly} \\
\mathcal{S}(e, i) = e & \\
\mathcal{I}(e, i) = e &
\end{array}
$$

It is clear that in order to keep the generated message compliant with the grammar, the only possible randomness in the evaluator E are the functions $\mathcal{C}$ and $\mathcal{R}$.

5.4.2 Merging Messages Evaluator

A more complex Evaluator where a message is generated out of the composition of several messages (also maintaining its compliance with the underlying grammar) is showed here. This evaluator is illustrative and will be mentioned in future applications described along the paper. We begin by defining the following function.

Definition 2. Assuming that M represent the set of messages compliant with the grammar (e.g. those that may had been generated by the Fuzzer Evaluator), and P is the set of all possible *reduction paths* from all the expressions presented in such messages, the function ρ of the form

$$\rho : M \times P \rightarrow e \cup \{\varnothing\}$$

obtains, if success, the corresponding expression for the *reduction path*, $xs \in P$, starting from the root expression of the message $m \in M$ that

$$(root(m), xs) \Rightarrow_F^n (e, [])$$

otherwise, if none expression exists, it returns $\varnothing$.

As a consequence, to allow the generation of messages out of the composition of others, the environment state of E is defined to be $\theta = M \times P \times P$. The variables $\tau, \delta \in P$ will represent the *reduction paths* from the initial and last triggered rule respectively. Assuming the variables ψ and ξ to be like:

$$\psi = \delta \quad \vee \quad \psi = \tau +\!\!+ \delta$$

$$\xi \in \{e \mid \exists\, m \in \omega : e = \rho(m, \psi) \wedge e \neq \varnothing\}$$

the definitions of the inner functions are detailed below.

$$
\begin{array}{ll}
\mathcal{T}(e, \omega, \delta, \tau) = (\xi, \omega, \delta \lhd \xi, \tau) & \\
\mathcal{N}(e, \omega, \delta, \tau) = (e, \omega, e, \tau \lhd \delta) & \\
\mathcal{C}(e_1/../e_n, \omega, \delta, \tau) = i & \text{where } 1 \leq i \leq n \text{ and } e_i = \xi \\
\mathcal{R}(e^{(i,j)}, \omega, \delta, \tau) = k & \text{where } 0 \leq i \leq k \leq j \text{ and } k = length(\xi) \\
\mathcal{S}(e, i, \omega, \delta, \tau) = (\xi, \omega, \delta \lhd i, \tau) & \\
\mathcal{I}(e, i, \omega, \delta, \tau) = (\xi, \omega, \delta \lhd i, \tau) &
\end{array}
$$

Is it worth noting that when replacing the expression e by ξ, the underlying grammar is still matched, because ξ is reduced by the same rule. However, the value of ψ chosen will define a degree of fuzziness in the resulting message due to the complete or relative path location of the expressions.

5.5 Learning Techniques

A much more challenging issue is however to learn from observed messages and use this knowledge as a base of smart fuzzing.

To illustrate the importance of such technique, we consider the following toy grammar and use the evaluator previously described in section 5.4.1 to generate fuzzed fields.

```
username = alphanum *(alphanum/"-"/"_"/"%"/"&")
alphanum =  ALPHA / DIGIT
ALPHA    =  %x41-5A / %x61-7A        ; A-Z / a-z
DIGIT    =  %x30-39                  ; 0-9
```

It can be assumed that the priority by which items appear in an *username* consists in letters, number and then special symbols. However, a possible reduction of such grammar may look like:

$$username \rightarrow \text{d-\&\%\%3\&\%-\&q}$$

For this example, the evaluator is up to generate an *username*, and when reaching the second item of the sequence, it has to decide among five choices, giving a low priority to numbers and letters.

For this cause, the two interfaces below had been defined to provide some methods to generate "smart" evaluators:

- Record Choice Indexes
- Record Repetition Lengths

Both interfaces receive as input the *sequence reductions* from the first rule, allowing to record statistics of repetition length and chosen items according to the function. Note that only these two methods are sufficient, because they are the only ones that can modify the evaluation flow of compliant messages.

6. STATE TRACKING

The Protocol State Evaluator is used to provide the evaluation of the fuzzing process. It uses the Protocol State Machine for two main tasks:

- identification of possible invalid transitions that were committed
- drive the fuzzer scenario on the next transitions to follow.

The Protocol State Evaluator described in this section is targeted at the SIP protocol. Domain specific knowledge is needed for this issue due to the fact that the evaluator has to be able to distinguish between correct or incorrect behavior.

6.1 Learning the Protocol State Machine

The Protocol State Machine on which the evaluator relies can be provided in two ways:

- a fully detailed state machine as specified by the standards
- a state machine induced from a sample of messages.

The latter approach was chosen in this research since different implementations do not really work in the same manner. Even if a device does not behave as expected by the protocol specification, this does not mean that a vulnerability was found. Therefore, in order to evaluate the impact of the crafted input, the normal behavior of the target entity should be known a priori.

SIP messages follow a hierarchy where Dialogs and Transactions are identified during a session. A dialog is uniquely identified by the *Call-ID* and a local and remote tag; such tags are presented in the *From* and *To* headers. Meanwhile, a transaction is identified by the *CSeq* header and the *Via Branches* of the top most *Via* header located in the message. Thus, a transaction belongs to only one dialog, but the latest may have many transactions. Also, a dialog is kept between two entities, even in the case where more entities are involved in the session.

To assume a simplistic model, the state machine is only for transactions rather than dialogs. Intermediate transactions may arise at specific states leading to a final state.

Figure 4 illustrates a simplified state machine that could have been obtained from samples of INVITE transactions.

6.2 Testing and Simulating an Entity

The Testing State Machine of the protocol is used to guide the testing by emulating malicious or normal behaviors. Such a state machine follows the principles of the Event-driven Extended Finite State Machine (EEFSM) described by David Lee et al. in [10]. However, in our approach, the above algorithm may not only be used to follow the system state but also to simulate and force one entity to perform different assessment operations.

The evaluation process is as follows: once a message is received and analyzed, the state machine should decide which will be the new outgoing message. This decision will be send to the Fuzzer Evaluator in order to generate the requested message. Meanwhile, the Protocol State Machine obtained by the sample data searches all the messages that match the following conditions

- the type message received is equal to the found messages in the sample (i.e. the message type may be the request methods or the reply values as *INVITE, CANCEL, 180 RINGING, 200 OK, etc*)
- the method *CSeq* in the message is equal to the one in the samples. The second item is mostly relevant to the messages that are replies, for example, the *200 OK* works for several type of messages, but the *CSeq* method defines to which type of method is replying.

Based on this process, the Protocol State Machine reports the existence of abnormal behaviors (e.g. unknown transitions).

The variables presented in each transition of the state machine correspond to the *local and remote Tag* and *Call-ID* (defining a Dialog) and the *CSeq* and *Via Branches* (defining a Transaction). The ones defining the Dialog will be invariant during the following. The one referring to the Transaction may change in the case of an interrupting or following transaction.

Note that the emulated behavior of the Protocol State Evaluator may change from one transaction to another. This is happening, because in a same Dialog different transaction may be initiated by any of the entities. The only thing left is the randomness to initiate a new transaction as well as the randomness to select among the possible set of message types to be send.

7. EXPERIMENTAL RESULTS

We followed the Iana[2] standard parameters supporting all the 43 SIP related RFCs mentioned in the document. This lead us to a fuzzer grammar composed of:

Class	Occurrences
Terminals (fixed strings and Character classes)	524
non-Terminals	591
Choices	231
Sequences	498
Repetitions	307

The fuzzer can be configured to respond in different ways according to its defined interfaces. Different types of behaviors have been tested using the infrastructure defined in section 5 which gave different scenarios behaviors.

The first, simplest and less effective, is the random generation of the whole message. In a probabilistic manner, such messages are compliant with the syntax provided by the protocol, but in fact, as was mentioned in section 5.5, they mostly are garbage that will be in most cases ignored by the target entity.

The second approach which results in more interesting messages consists in the mutation of existing messages. This mutation will tamper a set of fields in the original message based on a probabilistic mechanism. These latter values may be obtained by other rules, by other choices or with the result from a specific function defined by the interface.

The third scenario consists in learning from existing messages the different manners and probabilities in which rules may be reduced. Thus, the probability is estimated according to the global path starting from the root rule or from the path rooted in the current rule. Once the messages are

[2]http://www.iana.org/assignments/sip-parameters

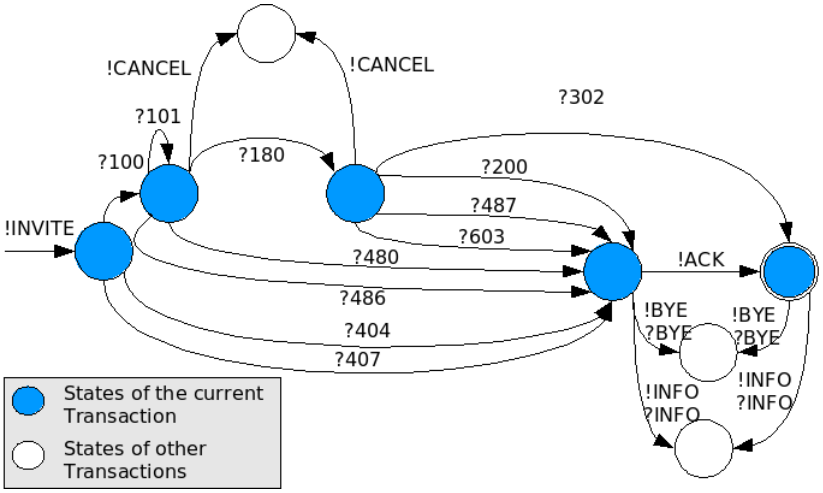

Figure 4: Learned State Machine from an INVITE message
The transition with quotation marks (?) means that these messages correspond to the responding entity. The exclamation mark (!) means that the messages belong to the requesting entity.

learned, any of the two previous techniques may be used. Note that as the entity has some knowledge now, the first technique will also result in messages similar to normal ones

Finally, the last technique comes from the idea proposed by Lin Li et al. [11]. They described a self-assembly process for software components. A component is composed by a series of inputs and outputs, such that in order to assembly components their input/output have to be considered as if they were molecules. The assembly process is an analogy with real molecules, where the pressure, temperature and area to promote the assemblies are modeled. Thus, in this article the idea of assembly fields of messages depending on their inputs and output is considered. The input of each field may be the global path from the root rule or the path from the current rule. While the output are all the items that may be reduced in the current field (i.e. Terminals, Non-Terminals, Choice Index, Number of Repetitions, etc). In this way, one field is attached to another, generating a message that is a composition of other messages. The function that defines which of the inputs may match with which output is left to the interface defined by the fuzzer. In this way approaches for Genetic Algorithms (GA) may be defined.

7.1 State Fuzzer Effectiveness

The evaluator may evolve as well. Different environments may be defined based on the sequence of messages to test. Such sequences may be acquired from the sample sessions or manually constructed. It is important to specify which messages in such sequences are the ones that should be fuzzed, which is the time range in which each type should be responded, the behavior after unknown transitions and the type of messages that may be the appear as a response.

7.2 Tested Scenarios

We tested our approach on different SIP enabled equipments used in real networks: our testbed consists in a Cisco 7940G, Cisco Call Manager 5.1, Thomson ST2020, Thomson ST2030, Linksys SPA941, Grandstream GXV-3000, Grandstream BudgeTone 200 and Asterisk/OpenSer proxies.

We manually constructed a set of scenarios that were applied to the equipments. These scenarios consisted in a INVITE transaction followed by a CANCEL transaction as illustrated in figure 5. Also not described in the latter figure, INVITE transactions that requires authentication. Register transactions were also taken in account to test devices either simulating to be an end point entity or a Registrar.

We tested and compared our results with the well known Protos [14] suite. Using our fuzzer we found 0-day vulnerabilities in all of the devices. Some of the stateless vulnerabilities found (i.e. those that require only one message to lead the device to an inconsistent state) were found in the first message generated for the scenario while others were obtained after a longer interaction. We observed that even in the latter case, if that messages are send at anytime, the device is not able to handle them. Among the most frequent vulnerabilities we found string overflows, buffer overflows and remote code execution vulnerabilities. We discovered multiple “Invite of Death” messages capable to kill one device with one single packet

Some of the tested equipment was really robust in term of input handling: the only vulnerabilities found were stateful, where a sequence of messages with particular fields and a specific order had to be send in order to take down the device. Figure 6 shows one of such stateful vulnerability found, where the messages are all syntactically compliant with the protocol, but their sequence order in the transaction and dialogs provoked a DoS in the target device. The messages are illustrative, but keeping them anonymous allowed the vendor to fix the bug. We have discovered even sequences of up to 10 messages leading to the crash of a hardphone. Such sequences can not be obtained by simple stateless fuzzers, showing us the interest of stateful SIP fuzzers. We have informed all the vendors and left them the necessary time to fix the vulnerabilities. We will release shortly on the usual dissemination channels the security advisories and the proof of concept exploit code.

Two vulnerabilities that we are allowed to present in this paper, have been disclosed by us on the voipsec and full-

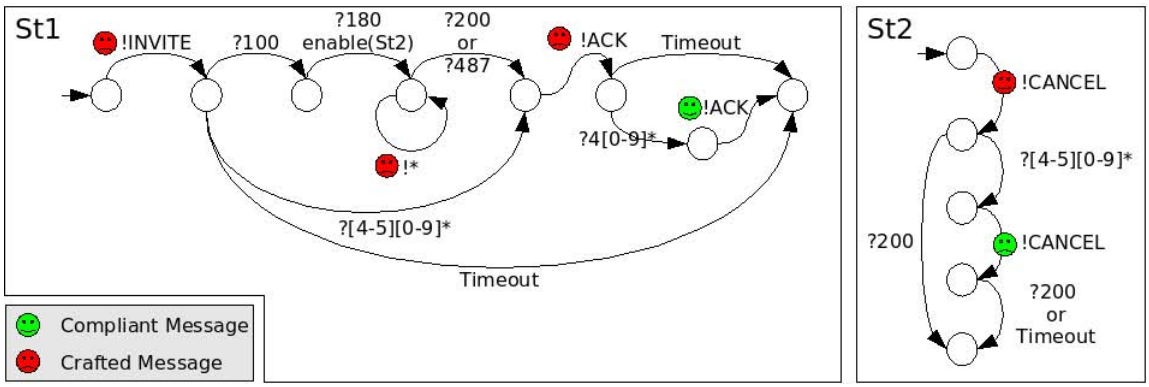

Figure 5: Statefull SIP Scenario

The quotation (?) and exclamation marks (!) are as in Figure 4.
The *enable()* function creates a fork in the current state machine to enable the other state machine, meanwhile, information like Dialog ID and Transaction ID is transfer to keep the state awareness.

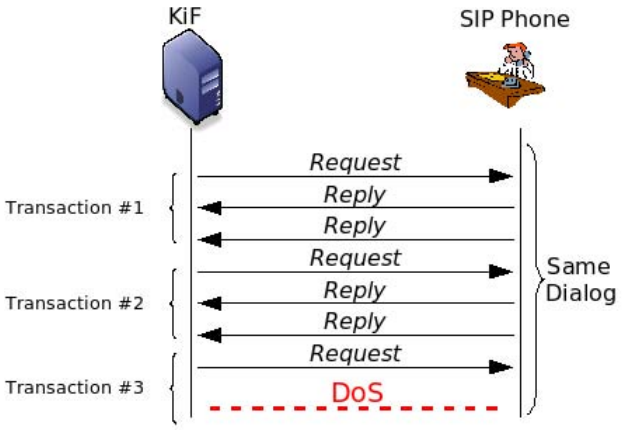

Figure 6: Statefull Vulnerability Example

disclosure mailing lists one day before this paper was submitted. The vendors confirmed them and also provided fixed software for these issues.

- Asterisk: after sending a crafted message the software crashes abruptly. The message in this case is an anonymous INVITE where the SDP contains 2 connection headers. The first one must be valid, however the second should not having an invalid IP address. The callee needs not to be a valid user or dialplan. In case where Asterisk is set to disallow anonymous call, a valid user and password should be known, and while responding the corresponding INVITE challenge the information should be crafted as above. Asterisk 1.4.1, 1.2.14, 1.2.15, 1.2.16 versions are affected by it.
- Cisco Phone 7940 running firmware P0S3-07-4-00: after sending a crafted INVITE message the device immediately reboots. The phone does not check properly the sipURI field of the Remote-Party-ID header in the message.

The previously mentioned vulnerabilities are particularly dangerous since one packet is capable to take down either a Asterisk PBX or an individual phone. In the case of the Asterisk PBX, the vulnerability can be used to execute remote code and take control of the PBX leading thus to major abuses, among which fraudulous service usage, voice eavesdropping and a further penetration of the internal network are the leading ones. On the other hand the taking down of remote phones with a single message is also very dangerous in a pure VoIP network. Along the overall tests we also learned which fields were ignored by each entity, which fields were used without any type of processing, which helped us to improve our fuzzing techniques. However, many entities generate messages that syntactically incorrect after some crafted messages rather than just ignoring or rejecting them.

8. CONCLUSIONS

Our paper describes a stateful protocol fuzzer for SIP. The main contribution of our paper is a flexible, adaptive fuzzer capable to track the state of the targeted application and device. One of the components of our work is quite generic and reusable for any protocol for which an underlying grammar is known. The second one is dependent on the domain specifics (SIP). To the best of our knowledge, this is the first SIP fuzzer capable to go beyond the simple generation of random input data. Our method is based on a learning algorithm where real network traces are used to learn and train an attack automata. This automata is evolving during the fuzzing process. We performed tests on VoIP phones and the results are promising: for each phone we found at least one vulnerability and several protocol errors. We follow a responsible disclosure policy, where vendors were informed and left the time to fix the issues. We could detail in this paper only the vulnerabilities, where the vendors could fix the concerned software/firmware, but in the short future all will be announced over the usual dissemination channels and on our website[3]. We will continue our work by integrating other protocols (H.323., RTP), testing more devices (session border controllers, routers, media gateway controllers) and refining the learning/testing algorithms used in our frame-

[3]http://madynes.loria.fr/

work.

9. REFERENCES

[1] H. Abdelnur, R. State, I. Chrisment, and C. Popi. "Assessing the security of VoIP Services". In *The 10th IFIP/IEEE Symposium on Integrated Management (IM 2007)*, Munich, Germany, May 2007.

[2] D. Aitel. "The Advantages of Block-Based Protocol Analysis for Security Testing". Immunity Inc, February 2002.

[3] D. Aitel. "MSRPC Fuzzing with SPIKE 2006". Immunity Inc, August 2006.

[4] G. Banks, M. Cova, V. Felmetsger, K. C. Almeroth, R. A. Kemmerer, and G. Vigna. Snooze: Toward a stateful network protocol fuzzer. In S. K. Katsikas, J. Lopez, M. Backes, S. Gritzalis, and B. Preneel, editors, *ISC*, volume 4176 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2006.

[5] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[6] D. Crocker. "Augmented BNF for Syntax Specifications: ABNF". Standards Track, November 1997.

[7] S. Embleton, S. Sparks, and R. Cunningham. "Sidewinder: An Evolutionary Guidance System for Malicious Input Crafting". Black Hat, August 2006.

[8] B. Ford. "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation". Symposium on Principles of Programming Languages, January 2004.

[9] R. Kaksonen. "A Functional Method for Assessing Protocol Implementation Security", Licentiate Thesis. VTT Publications 447. ISBN 951-38-5873-1, 2001.

[10] D. Lee, D. Chen, R. Hao, R. Miller, J. Wu, and X. Yin. "A Formal Approach for Passive Testing of Protocol Data Portions". In *ICNP '02: Proceedings of the 10th IEEE International Conference on Network Protocols*, pages 122–131. IEEE Computer Society, 2002.

[11] L. Li, N. Krasnogor, and J. Garibaldi. "Automated self-assembly programming paradigm: Initial investigations". In *The Third IEEE International Workshop on Engineering of Autonomic and Autonomous Systems*, pages 25–34, Potsdamn, Germany, 2006. IEEE Computer Society.

[12] P. M. Maurer. "Generating Test Data with Enhanced Context-Free Grammars". *IEEE Softw.*, 7(4):50–55, 1990.

[13] H. Sengar, D. Wijesekera, H. Wang, and S. Jajodia. Voip intrusion detection through interacting protocol state machines. In *DSN*, pages 393–402. IEEE Computer Society, 2006.

[14] O. University. PROTOS Test-Suite: c07-sip. http://www.ee.oulu.fi/research/ouspg/protos/testing/c07/sip, 2005.