Fuzz-testing complex protocols

Iustina Melinte
Computer Science and Engineering Department
University POLITEHNICA of Bucharest
Bucharest, Romania
{iustina_camelia.melinte}@cti.pub.ro

***Abstract*—Fuzzing has proven to be an effective method to determine the security of implementations with regard to malicious input. There are many approaches in this area, covering all testing strategies - white-box, grey-box and black box - and providing the freedom to combine methods and tools, up to the imagination and knowledge of the tester.**

In this paper we identify some of the features that determine the flexibility and effectiveness of a fuzzing framework. Our focus is oriented towards multi-protocol network fuzzing, grey-box and model-driven, able to handle stateful protocols. We present a case study on the IKE version 2, a security-oriented protocol with a high degree of complexity.

***Keywords*-network protocol fuzzing; IKEv2; model-based fuzzing, stateful protocols**

I. Introduction

For network protocol fuzzers, usually a formal model is given in the form of a RFC (request for comments). These specifications have a high level of abstraction and might not provide enough information on how to handle unexpected events. This aspect usually becomes the responsability of the programmer that does the implementation. Vulnerable software may also be the result of misunderstood specifications or ambiguity in protocol model, which is very common in RFCs.

We will begin by describing the existent fuzzing approaches and the features that influence the flexibility of a fuzzing framework to support more complex protocols. In Section 2 we discuss the related work that has been done in the area of network protocol fuzzing, including the research on IKE. Section 3 presents a case study on IKE version 2. The conclusions and future ideas are discussed in Section 4.

A. Fuzzing methods and strategies

Fuzzing has many advantages over the traditional vulnerability testing metodologies, such as auditing code. One important aspect is the fact that it can have a high degree of independence of the internal complexity of the tested system. There are several strategies that prove the flexiblilty of fuzzing and which are differentiated by their level of understanding of the target protocol.

The simplest form is the black-box approach. This is also called *capture-replay* or *mutation* fuzzing because it uses messages from normal conversations which are then modified and send to the target system. The fuzzer is usually acting as a *proxy* or *man-in-the-middle*. This approach has a big disadvantage though: its effectiveness is highly dependent on the variety of the inputs, which might not cover all the possibilities described in the protocol specification.

At the opposite side is the white-box fuzzing, which relies on dynamic test generation.

The most flexible fuzzing strategy is still the grey-box one. In this approach, a formal model of the protocol is usually available and used for input generation (from which the names of *generation-based* and *model-based* fuzzing come). The model may be provided by the standard, the RFCs or even by UML sequence diagrams, as in [1]. No source code access is needed, but the fuzzer may have access to the target logs and may be able to monitor the process in order to detect when the process crashes.

B. Criteria

In order to achieve the degree of flexibility and protocol independence that we aim at, we need to identify the criteria that define a good fuzzing framework. By studying complex network protocols, we can discover the needs that should be met by such a tool, and generalize them in order to allow code reuse when fuzzing any kind of protocol. In the following paragraphs, we enumerate some of these criteria.

- Modularity and portability. The architecture is a critical factor that influences flexibility and code reuse. It should be easily extensible, with well defined (communication) interfaces, to allow the easy addition of new protocols. A discussion about the architecture of a fuzzer is presented in [2]. The programming language is relevant for this criteria and Python is a good example, used in many such tools.
- Protocol model description. When considering the model-based approach used in grey-box fuzzing, there is a need to describe the message sintax, as well as the states and the transitions between them, for stateful protocols. The use of a portable format, such as XML, is recommended, but the key point here is to provide a syntax that will accommodate even the most complex protocols. The needs can be identified by looking at different protocols, how they are constructed. Some examples are: dependencies between fields/payloads, encodings, automatic length and checksum calculation.

- Fuzzing algorithms and heuristics. These two refer to how and when the input is mutated. Specifically, the fuzzing functions usually implement heuristics that target specific vulnerabilities related to memory corruption, by inserting boundary values, zero or negative numbers, format strings, directory traversal and so on. These functions are also called *fuzz operators*. The effectiveness of the fuzzer is critically dependent on the engine that does the actual mutations. This component should have the strategy for traversing the state machine nodes (for example, simulation of invalid transitions, prioritization) and should be easily extended (wordlists, for example).
- Target monitoring. It's important to receive feedback on the target state after each fuzz is applied. Monitoring methods include capturing packets, watching the target process for memory errors and system logs. There should also be a mechanism to reset the target to a clean configuration.
- Results reporting. This aspect is relevant for the efficiency of the analisys that is done after the testing. This process can be simplified by detecting patterns in responses, and finding a simple way to visualize and compare tests in one glance.
- Utilities and helpers. Some examples of these are: crypographic module for security-oriented protocols, parsers for different formats and packet captures, encoders/decoders.

II. Related Work

In this section we will present some of the previous work that has been done in the area of network protocol fuzzing. SNOOZE [3] is one of the most noticeable fuzzers that provides support for stateful protocols, an essential feature that lacks in most of these tools. The authors used a model-based approach for testing SIP, describing its syntax through XML files. A similar approach to stateful fuzzing is called *behavioral*, as described in [1], using UML sequence diagrams for determining sequences of actions that lead to insecure states.

The block-based approach has been used by many fuzzers along the history, for example Sulley [4], SPIKE [5] and Autodafe [6]. Sulley is one of the fuzzing frameworks able to test complex protocols. The model is described through internal structures which include facilities for using automatic length calculators, checksum generators, dependencies and encoders. It has a modular architecture, process and network monitoring tools which are able to determine exactly the effect of each fuzzed message sent to the target, its causes and consequences. The main drawback of Sulley is the lack for stateful fuzzing (sending unexpected messages in the FSM, for example). Although it can describe valid transitions between states, the algorithm traverses these state-nodes in order, for each path between the start and the end state in the graph. Complex protocols usually have a high degree of dependency between adjacent states, and Sulley does not take into consideration the response from the target in order to determine the next state. For message fuzzing, Sulley uses an internal stack with named fields and blocks, which allows easy access to individual items, a very useful feature.

Both Sulley and SNOOZE use the same approach for applying mutations, specifically one that is based on the field type (integer or string). This might be a limitation though, because it does not allow flexibility in choosing which fuzzing function (and set of values) to apply to a certain field or block.

Since our case study is about IPsec, specifically IKE, we also researched the work that has been done in this area. In [7] and [8], Tsankov et al. tested the Openswan suite, using the actual implementation as a low level model for valid inputs generation. The setup that they used is modular, decoupling the generation and mutation phases. The tool, called SECFUZZ, overcame the difficulty of mutating encrypted messages by reading the Openswan log files, which provided the keys and all the cryptographic information of an IKE session.

PROTOS, a well-known network protocol fuzzer developed at the University of Oulu, also includes tests for fuzzing IKE [9]. The results prove its efficacity but unfortunately the tool is not public. At the same unversity previous research has been done with a tool called IKEFUZZ [10]. The complexity of PROTOS and IKEFUZZ seems to be far greater than SECFUZZ, and the two also use a model-based fuzzing approach.

From the above mentioned fuzzers some tendencies can be noticed, specifically the use of high-level languages due to the flexibility and the useful libraries that they can provide (Sulley, SECFUZZ, SNOOZE used Python) and the fact that the effectiveness of the fuzzer is directly proportional with its level of understanding of the model.

III. IPsec case study

The IPsec protocol has a high degree of complexity and its specification is written in many RFCs, with much attention given to security-oriented details, with the purpose to avoid any misunderstanding that could lead to insecure implementations. We focused only on the initialization phase of IPsec, which is the Internet Key Exchange.

Our case study was made on a IKEv2 implementation, Strongswan specifically. The approach was to use a model of the protocol, as described in RFC 5996 [12]. We extended the Sulley fuzzing framework and used its fuzzing engine and the network monitoring tool. The exchanges and messages syntax were also described using Sulley blocks, primitives and the state graph.

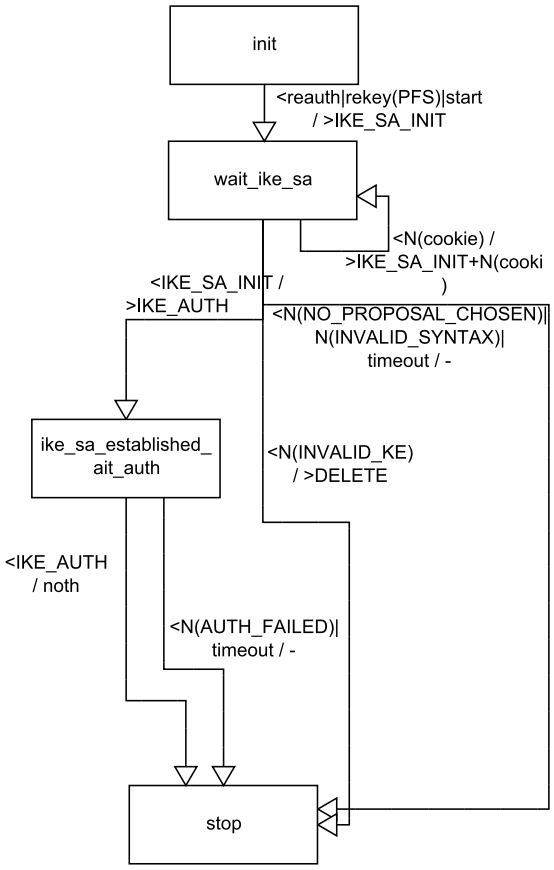

Figure 1. IKE SA state machine, initiator role

We focused our study on the phase 1 exchanges and only for the initiator role. The IKEv2 state machine that we used is depicted in Figure 1. There are 3 types of messages: IKE_SA_INIT, IKE_AUTH and DELETE, differently structured in payloads. The transitions are represented in the *<event>/<action>* form. The *events* are usually the receival of a message from the responder endpoint (the '<' symbol) or a local condition, such as process startup or a triggered rekeying phase. *N(..)* is the notation for the Notification message type. The *action* is usually sending a message (the '>' symbol). The 2 states that we targeted are the IKE_SA_INIT and IKE_AUTH. IKE_SA_INIT establishes the cryptgraphic material for IKE SA (Security Association). IKE_AUTH does the same but for the IPsec SA and also authenticates the peers.

The structure of the 2 message types that we used for the first 2 exchanges is presented in a formal way below. The notation $\{m\}SKx$ denotes that payloads are encrypted and integrity protected using some of the generated keys. The *sk(A)* is the digital signature of the agent *X* on the message *m*.

```
IKE_SA_INIT
A!B : HDR,SAi1,KEa,Na

IKE_AUTH
A!B : HDR,{IDa,[CERT,] [CERTREQ,]
[IDb],AUTHi,SAi2,TSa,TSb}SKa

where:
AUTHi={SAi1,gxA,Na,Nb,prfSKpi(IDa)}sk(A)
AUTHr={SAr1,gxB,Nb,Na,prfSKpr(IDb)}sk(B)
```

The architecture of the actual fuzzer is depicted in Figure 2.

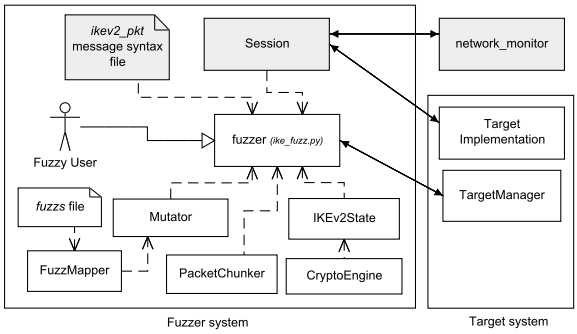

Figure 2. Architecture

The protocol messages, payloads and fields are described in the *ikev2_pkt* component, using the Sulley block-based syntax. The code below shows the description for the generic header, used for every type of payload defined in the rfc. The *s_byte* and *s_bits* define numeric fields, and *t_size* is a length field that automatically adjusts its size based on the content lenght of the *payl_name* parameter.

```
1 def generic_hdr(np,payl_name):
2     if s_block_start("gen_hdr_"+payl_name):
3         s_byte(np,format="binary",name="gen_hdr_"+
              payl_name+"_np",fuzzable=False)  #
              Next Payload
4         s_bits(0b00000000,8,name="gen_hdr_"+
              payl_name+"_reserved",fuzzable=False)
              # Critical+Reserved
5         t_size(payl_name, length=2, endian=">",
              name="gen_hdr_"+payl_name+"_len",
              fuzzable=False) # Payload Length
6         s_block_end("gen_hdr"+payl_name)
```

The second input of the fuzzer is the *fuzzs* file, which contains the test suite for a given state. This is in fact a list of fuzz operators that are applied in order, one for each protocol execution. A fuzz is made of a list of block or field names that are to be fuzzed, the fuzzing method, its parameters and a short description that will appear in the report. The strategies used for fuzzing can be applied at 3 levels: state machine fuzzing (invalid transitions in the FSM graph) message fuzzing (inserting random payloads, multiplying, scrambling) field fuzzing (boundary values for integer types, random strings or any other value that could exploit a memory vulnerability)

We only implemented the message and field fuzzing, and the methods that we used are:

- *in_list*: uses a list of values that will be tried, both valid (rfc compliant) and invalid
- *near_value*: uses a list of numbers that will be added to or substracted from the original integer value; applied mostly on length fields
- *sulley_default*: checks boundary values for integers, format strings or random values; this is implemented in Sulley
- *payload_multiply*: multiplies or deletes a certain payload
- *payload_scramble*: randomly reorders the payloads specified

The above fuzzing methods are implemented in the *Mutator* module, which applies one fuzz per protocol execution. This means that depending on the state that is to be fuzzed, all the valid messages before the fuzzed one will be sent intact, in order to bring the target in the desired state. This has the purpose to ensure code coverage.

The *PacketChunker* component works with raw packets. It extracts and replaces fields from sent and received messages in order to collect and set the information relevant to the state machine. It identifies these fields by searching for patterns.

Since IKE is a security protocol, it provides support for many cryptographic algorithms, hash functions and similar mechanisms. These are relevant for reaching a more deep state whithin the FSM, from the fuzzer perspective. The *CryptoEngine* component was made specifically for this purpose. It encrypts and authenticates messages in the IKE_AUTH state.

The *IKEv2State* holds the cryptographic material used by the *CryptoEngine* and additional information relevant to the state machine.

The fuzzing engine of the framework is implemented by the Sulley *Session* class. It traverses the state nodes in the graph, exchanges messages with the target and communicates with the network monitor component to determine the reactions of the target to its fuzzs.

The fuzzer is started by the *ike_fuzz.py* component. After each fuzzed message or sequence of messages, the *TargetManager* resets the state and configuration of the process.

Since Sulley lacks support for stateful protocols, we needed to add this functionality in order to allow transitions between states where messages are highly dependent on the information contained by the previous ones.

IV. Conclusion and Further Work

In this work, we considered the problem of testing complex network protocols through fuzzing. We identified some of the characteristics that can determine the efficacity and flexibility of a fuzzing framework. Our case study was focused on the IKE, a highly complex security-oriented protocol.

In the future, we plan to recreate some of the components of the framework according to the characteristics and architecture that we identified. For example, the fuzzing engine could use more flexible internal representations of the protocol data than the ones used by Sulley. The fuzzing algorithm should also provide support for stateful protocols by default (we added this functionality using an extension). We could also improve the Mutator component by adding more fuzzing methods, such as sending unexpected message types for state machine fuzzing.

We also intend to continue fuzzing the IKEv2 protocol, adding support for state machine fuzzing and the CREATE_CHILD_SA exchange.

Finally, any future work will take into consideration the reusability of the framework across other network protocols.

Acknowledgment

This work has been made in collaboration with Ixia company, Bucharest, Romania. The author would like to thank Octavian Radu, Dragos Comaneci and Razvan Rughinis for their time and support and for providing guidance and advice during this research.

References

[1] M. Schneider, J. Großmann, I. Schieferdecker, *Online Model-Based Behavioral Fuzzing*. Available: http://www.spacios.eu/sectest2013/pdfs/sectest2013_submission_9.pdf

[2] "Delivering malformed data for fuzz testing to software applications", http://www.google.com/patents/US8336102

[3] G. Banks, M. Cova, V. Felmetsger, K. C. Almeroth, R. A. Kemmerer and G. Vigna, *SNOOZE: Toward a stateful network protocol fuzzer* in ISC, 2006, pp. 343–358. Available: http://nmsl.cs.ucsb.edu/papers/139.pdf

[4] Sulley fuzzing framework, https://github.com/OpenRCE/sulley

[5] "SPIKE", http://www.immunitysec.com/resources-freesoftware.shtml

[6] M. Vuagnoux, *Autodafe: an Act of Software Torture* . Available: http://autodafe.sourceforge.net/docs/autodafe.pdf

[7] P. Tsankov, M. Torabi Dashti, D. Basin, *In-depth fuzz testing of IKE implementations*. Available: ftp://ftp.inf.ethz.ch/doc/tech-reports/7xx/747.pdf

[8] P. Tsankov, M. Torabi Dashti, D. Basin, *SECFUZZ: Fuzz-testing Security Protocols* . Available: http://www.infsec.ethz.ch/research/publications/pub2012/ike.pdf

[9] "Protos test-suite: c09-isakmp", https://www.ee.oulu.fi/research/ouspg/PROTOS_Test-Suite_c09-isakmp

[10] J. DeMott "ikefuzz", https://www.ee.oulu.fi/research/ouspg/ikefuzz

[11] "strongSwan", http://strongswan.org/

[12] "Internet Key Exchange Protocol Version 2 (IKEv2)", http://www.rfc-editor.org/rfc/rfc5996.txt