Fuzzing network security protocols

Iustina Melinte
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
{*iustina_camelia.melinte*}*@cti.pub.ro*

February 5, 2014

Abstract

Fuzzing is an effective testing method that can reveal vulnerabilities in software products by using unexpected inputs. We propose a gray-box, model-driven approach to fuzz testing the IKEv2 protocol, using a mutation-based algorithm with a wide variety of fuzz operators. The effectiveness of our approach is proved in a case study with various implementations of IKEv2.

Keywords - model-based fuzz-testing, gray-box, IKEv2

1 Introduction

Software testing is usually done by providing the application with inputs that are forseen in its specification. This provides little coverage, since test cases target a specific aspect of the implementation, anticipated by the programmer. Overcoming this limitation involves creating a wide variety of unexpected inputs. This is what fuzzing does.

There are many approaches to fuzz-testing, including black-box and gray-box (depending on level of knowledge of the target implementation), model-based and using mutated versions of various known valid inputs.

IPsec is one of the protocols that are used to implement Virtual Private Networks (VPNs). It provides security properties such as confidentiality, authentication and integrity protection.

IKEv2 (described in RFC 5996 [1]) is the protocol used in the initialization phase of IPsec, to establish cryptographic parameters for VPN connections. These are called security associations and define the encryption algorithms, key lengths, authentication methods and integrity protection functions that will be used for a session. The structure of an IKE message is defined through a payload list, each payload containing a list of fields.

There are two exhanges. The IKE_SA_INIT establishes a IKE security association and the IKE_AUTH phase authenticates the peer and establishes the IPsec security association.

Only the first exchange is unencrypted, which makes it more difficult to fuzz subsequent exchanges, due to the need to keep state information (cryptographic algorithms and their keys and parameters) for each test case or fuzz attempt.

In this paper, we propose a highly flexible, modular architecture for a fuzzer, given the criteria presented in [5]. Our approach is model-based, since we assume no knowledge about the tested implementation, only the specification (RFC).

Previous work has been done in the area of network protocol fuzzing. The model-driven and block-based approach was used in fuzzers such as Sulley [7] and SPIKE [6]. In [2] and [3], Tsankov et al. tested the Openswan suite, using the actual implementation as a low level model for valid inputs generation. Their tool, called SECFUZZ, overcame the difficulty of mutating encrypted messages by reading the log files, which provided the keys and all the cryptographic information of an IKE session. We consider that our model-based approach using a partial IKEv2 implementation within the fuzzer is more flexible and allows us to test any IKEv2 software.

The remainder of this paper is organized as follows. In section 2 we describe the general architecture for a network protocol fuzzer. Section 3 provides some details about certain components, such as model description, fuzzing heuristics (operators), target management and reporting processes. The setup that we used for our testing is provided in the fourth section. We discuss the discovered vulnerabilities in section 5. The last section concludes this paper and provides some ideas for future development.

2 Architecture

The fuzzer architecture is depicted in Figure 1.

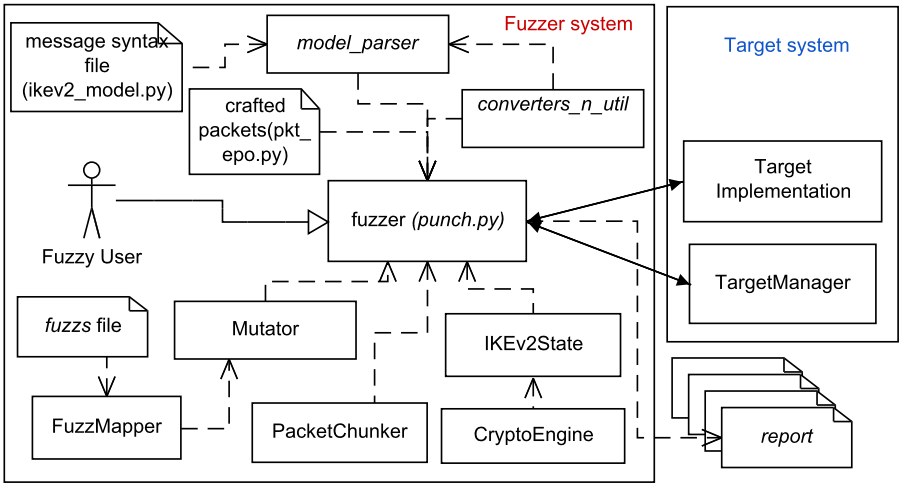

Figure 1: Fuzzer architecture

As we previously mentioned, we use the protocol model (described in RFC 5996 [1]) to generate the initial valid messages. The *message syntax file* contains a block-based description of the IKEv2 message types. This is the input for the *model_parser* module

which creates a tree-like internal structure containing the messages with a valid structure and valid field values. The Mutator applies the fuzzing operators based on the *fuzzs* input file, which is configured by the user. All the processes are handled by the central component, *punch*, which generates the valid inputs, applies one mutation at a time, sends the malicious packets to the target and then handles the response message if it is the case. This component synchronizes with the *TargetManager*, which resets the connections and the target configuration between fuzz-tests. Since all the messages after the initial exchange should be encrypted and authenticated, the *IKEv2State* component keeps track of the keys and algorithms negotiated for each session. The cryptographic operations are applied by the *CryptoEngine*.

3 Implementation

Having a modular architecture, like the one we've seen in the previous section, enhances flexibility and allows code reuse. For example, adding a new protocol would only require a new model description file, a new *fuzzs* list and maybe some customizations in the *Mutator* component.

3.1 Protocol model description

The syntax of the protocol is described in a block-based manner, similar to the one used by Sulley [7]. We use 3 types of primitives: message, block and field. Each block has a list of items that it contains. There are several types of fields, based on their representation. An example is given in Listing 1 for the Nonce payload in IKEv2.

Listing 1: Block description for IKEv2 Nonce payload

```
def nonce(np,nonce_data):
  if block_start("nonce"):
    generic_hdr(np,"nonce")
    field('hex',len(nonce_data)/2,nonce_data,name='nonce_data')
  block_end()
```

The syntax is flexible enough to allow the description of nested blocks (like the *generic_hdr* above), optional elements and dependencies. For each field, the parameters are *type, size* (measured in bytes), *value* and *name*, in this order. The name is used as a field identifier in the fuzzs list, along with identifiers for fuzzing operators and other parameters.

3.2 Fuzzing heuristics and operators

The fuzzing is done at four levels: field, payload, message and state machine. We use a different set of operators for each of these. Table 1 offers a description for all the fuzz operators that we used.

Table 1: Fuzz operators

Level	**Fuzz operator**	**Description**
Field fuzzing	value from list	uses a list of values that will be tried, invalid or valid (RFC compliant/with a significance) but in a different context (example: setting a value to 0); preserves the original length
	near value	(numeric fields) uses a list of numbers that will be added to or substracted from the original integer value; applied mostly on length fields; preserves the original length
	replace bytes at random position	uses a list of characters or character sequences to be inserted at a random position inside a certain field (for example null terminators, format strings and other special characters); preserves the original length
Payload fuzzing	delete from field	removes all fields from a payload, from the given one to the end; updates all length fields
Message fuzzing	multiply payload	multiplies or deletes a certain payload, updating length fields
	scramble payloads	randomly reorders the payloads specified
	insert payload	inserts an unexpected payload at a given position inside a message (for example, inserting a nonce payload in the IKE_AUTH exchange)
FSM fuzzing	insert informational message	notification/delete connection/configuration; inserted between IKE_SA_INIT and IKE_AUTH exchanges

The message types that are considered include IKE_SA_INIT, IKE_AUTH and informational.

As seen in Table 1, IKEv2 state-machine fuzzing can be done by sending various notification messages (there are many types listed in the RFC) or instructing the target to delete the security association, actions that may cause unexpected behaviours.

Due to the rigorous checking of the message sequence in this security protocol (and the few states) it wouldn't have brought much value creating a FSM description to use as input for state machine fuzzing. Instead, we described manually some 'dangerous' message sequences that could have a negative impact, but still passing the initial verifications and reaching a further processing point.

The main purpose of having so many fuzzing operators is increasing the coverage of our testing. There are also other criteria that determine the coverage, including the variety of the initial configurations and the way in which the fuzz operators are applied to fields. For the first one, an idea would be using different ID types (IPv4, IPv6, FQDN, ASN.1 X.500 Distinguished Name for example), Traffic Selector types and various security associations. A more advanced discussion about coverage is made at [4].

3.3 Target management

As we've seen in the architecture, there is a separate component that runs on the target and synchronizes with the fuzzer after each received message. It's main purpose is to monitor the process/daemon and to reset the peer to a clean configuration. We used a dynamic memory analysis tool called Valgrind [13] to detect memory faults.

3.4 Results reporting

The reporting process influences the effectiveness of results analysis. The fuzzer generates detailed raports that allow us to determine with great precision the messages that caused faults in the target, to easily reconstuct the fuzzing scenarios. It also uses a pattern detection mechanism to identify responses.

4 Experimental Setup

Our case study was made on several Linux IKEv2 implementations running inside multiple virtual machines. The fuzzer was run from the host operating system. Due to the fact that it is written in plain Python, the fuzzer could have been easily deployed in any operating system, provided that it had a working network communication link with the target.

The next steps included configuring the fuzzer and the *TargetManager* with the IPs and other parameters, copying the *TargetManager* in the virtual machine and then starting a new fuzzing session.

We used a simple configuration for the IPsec peer, with pre-shared keys as the authentication method and increasing the log level to help with debugging. An example is given in Listings 2 and 3, for *libreswan* IPsec implementation.

Listing 2: ipsec.conf

```
config setup
    ...
    plutodebug=all
    plutorestartoncrash=yes


conn myfuzz
    ikev2=insist
    auto=add
    type=tunnel
    authby=secret
    left=172.16.186.184
    right=172.16.186.1
    ike=3des-md5;modp1024
```

Listing 3: ipsec.secrets

```
172.16.186.184 172.16.186.1: PSK "youpskhere"
```

5 Scenarios and Results

We tested the fuzzer against libreswan 3.7, openswan 2.6.39 and strongSwan 5.1.0. These are the latest versions of the chosen implementations. Libreswan is a descendant (fork) of openswan 2.6.38.

We discovered some previously unreported vulnerabilities in libreswan (and openswan too). These were mostly related to the initial message from the IKE_SA_INIT exchange, allowing any unauthenticated party to cause the *pluto* IKEv2 daemon to crash and restart.

The IKE_SA_INIT message, as defined in RFC 5996 [1], contains a header and the following payloads, in this order: SA (security association), KE (key exchange) and nonce payload. Since all of these have a variable length, the receiver can identify and delimit them by reading the Next Payload field from the previous payload and their length fields. For example, in the initial exchange, the IKE header next payload field contains the identifier for the SA payload, the SA next payload is the KE identifier and so on.

Altering the Next Payload field while keeping the same message structure, in both the IKE_SA_INIT header and the KE Payload, causes the program to crash while trying to dereference a NULL pointer to the expected next payloads that were not identified. In particular, the first bug was caused by dereferencing a pointer to the a presumably received SA payload, and the second was for the Nonce payload.

Another bug was exposed by sending a IKE_SA_INIT message with exchange type set to 0, which also caused the *pluto* daemon to restart after a null pointer dereference.

These vulnerabilities affected libreswan version up to 3.7 and also all versions of openswan including 2.6.39. Our findings were reported to the developers of libreswan and openswan, and a security patch was released (see CVE 2013-6467 [12]).

6 Conclusion and Further Work

This paper presented a highly flexible, modular architecture for a network protocol fuzzer. Our approach was model-driven (using the IKEv2 RFC), with a block-based description of the messages syntax. Support for the cryptographic operations was included in the fuzzer implementation, allowing us to properly encrypt and authenticate the malicious messages that triggered further points in the protocol state machine. Testing coverage was achieved by using a wide variety of fuzzing operators, at different levels, to trigger unexpected behaviours in the target.

Future ideas include adding support for new protocols, which includes defining visible interfaces for communication, model descriptions and mutation functions. In the IPsec

IKEv2 case, we could improve the coverage of our testing by providing a wider variety of configurations and support for fuzzing the responder role.

Acknowledgment

The author would like to thank Paul Wouters and the other developers of libreswan for their involvement and responsiveness regarding the identified vulnerabilities, and Laura Gheorghe for coordination of this project.

References

[1] *Internet Key Exchange Protocol Version 2 (IKEv2)*, RFC 5996, 2010

[2] P. Tsankov, M. Torabi Dashti, D. Basin, *In-depth fuzz testing of IKE implementations*. Available: http://e-collection.library.ethz.ch/eserv/eth:5676/eth-5676-01.pdf (Last Accessed: January 2014)

[3] P. Tsankov, M. Torabi Dashti, D. Basin, *SECFUZZ: Fuzz-testing Security Protocols*. Available: http://www.infsec.ethz.ch/research/publications/pub2012/ike.pdf (Last Accessed: January 2014)

[4] P. Tsankov, M. Torabi Dashti, D. Basin, *Semi-valid Input Coverage for Fuzz Testing*. Available: http://people.inf.ethz.ch/torabidm/issta2013.pdf (Last Accessed: January 2014)

[5] Iustina Melinte, *Fuzz-testing complex protocols*. Available: https://github.com/iustinam90/buster/blob/master/_docs/Iustina_Melinte_FuzzTestingComplexProtocols.pdf (Last Accessed: January 2014)

[6] *SPIKE*, http://www.immunitysec.com/resources-freesoftware.shtml (Last Accessed: January 2014)

[7] *Sulley fuzzing framework*, https://github.com/OpenRCE/sulley (Last Accessed: January 2014)

[8] J. DeMott *ikefuzz*, https://www.ee.oulu.fi/research/ouspg/ikefuzz (Last Accessed: January 2014)

[9] *libreswan*, https://libreswan.org/ (Last Accessed: January 2014)

[10] *openswan*, http://openswan.org (Last Accessed: January 2014)

[11] *strongSwan*, http://strongswan.org/ (Last Accessed: January 2014)

[12] CVE 2013-6467, https://libreswan.org/security/CVE-2013-6467/ (Last Accessed: January 2014)

[13] N. Nethercote and J. Seward, *Valgrind: a framework for heavyweight dynamic binary instrumentation*. Available: http://valgrind.org/docs/valgrind2007.pdf (Last Accessed: January 2014)