To appear in:
In Proc. of the 7th International Workshop on Automation of Software Test (AST 2012).
June 2-3, 2012, Zurich, Switzerland.

SecFuzz: Fuzz-testing Security Protocols

Petar Tsankov, Mohammad Torabi Dashti, David Basin
Institute of Information Security, ETH Zurich
{ptsankov|torabidm|basin}@inf.ethz.ch

***Abstract*—We propose a light-weight, yet effective, technique for fuzz-testing security protocols. Our technique is modular, it exercises (stateful) protocol implementations in depth, and handles encrypted traffic. We use a concrete implementation of the protocol to generate valid inputs, and mutate the inputs using a set of fuzz operators. A dynamic memory analysis tool monitors the execution as an oracle to detect the vulnerabilities exposed by fuzz-testing. We provide the fuzzer with the necessary keys and cryptographic algorithms in order to properly mutate encrypted messages. We present a case study on two widely used, mature implementations of the Internet Key Exchange (IKE) protocol and report on two new vulnerabilities discovered by our fuzz-testing tool. We also compare the effectiveness of our technique to two existing model-based fuzz-testing tools for IKE.**

I. Introduction

Context. Software testing is an exploratory process that helps us to gain confidence in the correctness of a software system with respect to its specification. Testing is often performed by executing the system using the inputs that are foreseen in the specification. The system's output is then compared to the output prescribed by the specification to determine a pass/fail verdict. For testing implementations of security protocols, one must investigate the behaviors of the system also using inputs that are not foreseen by the specification. This is because the attacker may try *any* input in order to force the system into an insecure state. This is a serious problem in practice: of the top 25 most dangerous software errors listed in the Common Weakness Enumeration database [1], the top 4 are due to improper handling of unexpected inputs.

Fuzz-testing addresses this problem: the system is executed using the inputs that are not anticipated by the specification, and then the system's behavior is checked for failures [2], [3]. Unexpected inputs are often generated by mutating (or, fuzzing) valid inputs according to a set of fuzz operators. Since the unexpected inputs are usually not associated to any output in the specification, the specification cannot be used as an oracle to issue pass/fail verdicts. One can however look for the behaviors that are generally undesired in software systems such as accessing unallocated memory, and raise an alarm if such behaviors are witnessed. Various dynamic memory analysis tools can be used for this purpose.

Fuzz-testing stateful systems (such as security protocols) requires generating valid inputs that explore the system in depth. That is, one uses valid inputs to guide the system into states that are deep in its state space, and then feeds the system with unexpected inputs. Various sources can be used for generating valid inputs. One can gather typical input data (e.g. download image files from the Internet in order to test an image processing software), use a formal model of the system (known as model-based fuzz-testing), analyze the source code of the system (known as white-box fuzz-testing), etc.

Contributions. We use a concrete implementation of the protocol as a black-box for generating valid inputs. For example, to test the server side of a security protocol, we use the client's implementation to generate valid inputs for the server. The server and the client both execute the security protocol, with the provision that all messages sent to the server pass through our fuzzer. The fuzzer obtains valid inputs from the client without having to "understand" the underlying protocol. The fuzzer mutates the inputs according to a set of fuzz operators, and then sends them to the server. Note that we require that a running version of the "opponent" of the system under test is available. In the example above, the server is the system under test (SUT), and the client is the server's opponent. This requirement is often met when testing protocols. It is however not universally satisfied, e.g. think of Web services for which only a WSDL description is available. We remark that our approach is different from the aforementioned input generation methods as we do not use the model or the source code of the SUT to generate valid inputs for fuzz-testing.

By decoupling the input generation and the input mutation steps we gain modularity: we can use the same fuzzer to test different protocols. All we need to test the Alice of a protocol is a running Bob of the same protocol, and a set of fuzz operators (we come back to these later). This method is however not directly applicable to security protocols, because security protocols exchange messages that are encrypted and possibly contain randomly generated numbers. Since mutating encrypted messages almost always results in garbage, the fuzzer must have access to the encryption keys, random numbers used in encryptions, and the cryptographic algorithms used in the protocol. We solve this problem in our case study by configuring the system such that the fuzzer can read the encryption keys, pointers to cryptographic algorithms, etc. from a log file.

In addition to generating inputs that explore the implementation in depth, the effectiveness of fuzz-testing depends on the fuzz operators. The fuzz operators mutate valid inputs, ideally to the effect of exposing vulnerabilities. Here, we define three classes of fuzz operators for security protocols. We justify the choice of these operators by referring to common vulnerability databases and argue that our operators can expose prevalent software weaknesses. We show their effectiveness in practice through our case study. Moreover, in the context of our case study we compare our fuzz-testing tool to two model-based fuzz-testing tools, namely PROTOS [4] and IKEFUZZ [5]. The comparison shows that our tool is incomparable to PROTOS, and better than IKEFUZZ, in terms of the vulnerabilities the tools discover.

To summarize, our contributions are threefold. First, we propose a modular fuzz-testing technique for testing stateful security protocols that handles encrypted traffic. Second, we give a set of fuzz operators that are effective for finding vulnerabilities in security protocols. Finally, we empirically evaluate the effectiveness of our technique and the proposed fuzz operators through a case study on the Internet Key Exchange protocol (IKE). The results demonstrate that our approach can find real vulnerabilities: we report on previously unknown vulnerabilities in two widely used, mature implementations of IKE, namely OpenSwan and Shrew Soft's VPN Client. The fuzzer used in our case study, dubbed SECFUZZ, is implemented in Python and is publicly available at http://www.infsec.ethz.ch/research/software/.

Related work. White-box fuzz-testing [6]–[8] and model-based fuzz-testing [9]–[11] are related to our work. These two methods generate valid inputs for exploring the SUT in depth. What distinguishes these from our technique is that they assume access to, respectively, the source code and a model of the SUT. In white-box fuzz-testing, generating new valid inputs is expensive (in general, intractable), and constructing accurate models for model-based testing is a challenging task (e.g. see [12]). This makes our technique a light-weight alternative to these methods, as we make no assumptions on the availability of the source code or system models. White-box and model-based fuzzers do however not rely on any implementation of the opponent of the SUT. They are therefore potentially more thorough than our technique.

Outline. In Section II we present our modular fuzz-testing method, i.e. we explain how the SUT, the fuzzer, and the implementation used for generating valid inputs are executed together. In Section III we define our fuzz operators and argue that they are effective for exposing software vulnerabilities. In Section IV we present our case study on IKE and the vulnerabilities that we found in OpenSwan and the Shrew Soft's VPN Client. There, we also compare the effectiveness of SECFUZZ to two model-based fuzz-testing tools for IKE.

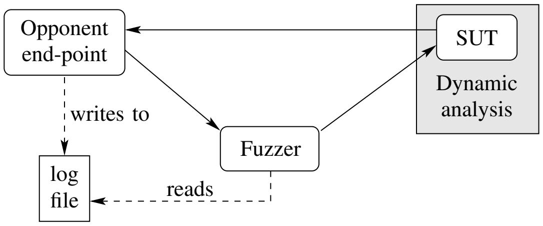

Figure 1. Test setup: messages sent to the SUT pass through the fuzzer. The input generating end-point shares information (e.g. keys, encryption algorithms, etc.) with the fuzzer using a log file.

II. SECFUZZ SETUP

Our aim is to test a security protocol implementation for failures by executing it using unexpected inputs. In this paper we use the term *failure* to indicate the manifestation of a fault, where a *fault* is the concrete incorrect step performed by the system. We achieve this in three steps: first, a concrete implementation of the protocol is used to generate inputs for the SUT; we refer to these inputs as *valid inputs*. Second, a fuzzer modifies the valid inputs; we refer to the modified inputs as *mutated inputs*. Finally, the SUT is executed using the mutated inputs and its behavior is checked for failures.

Figure 1 illustrates our test setup. The SUT is one of the end-points participating in a security protocol and the opponent is the other end-point. For example, if the SUT is the security protocol's Alice then the opponent end-point is the corresponding Bob. Note that the two end-points need not be "the same software", i.e. they may be different implementations of the same security protocol. In contrast to a standard deployment setting in which the two end-points communicate directly, in our test setup we configure the communication environment to route the messages destined for the SUT through the fuzzer. As a result, the opponent end-point generates and passes valid inputs to the fuzzer. The fuzzer's role is to mutate the valid inputs and send them to the SUT. Furthermore, we use a shared medium (depicted as a log file in Figure 1) to allow the opponent end-point to communicate data such as encryption keys to the fuzzer. The SUT is executed within a dynamic memory analysis tool which serves as an oracle for detecting failures. Note that our test setup is general, i.e. suitable for fuzz-testing any two-party security protocols, and its extension to n-party security protocols is straightforward.

We remark that the fuzzer needs "fresh" messages from an active session of the protocol. Unlike in simple protocols where the fuzzer can use previously captured inputs (as in, e.g., Codenomicon's traffic capture fuzzer [13]), the inputs to security protocol implementations use randomness, e.g. a fresh key, and hence cannot be reused. Our test setup addresses this by placing the fuzzer as a mid-point in the communication channel connecting the opponent end-point to the SUT.

In what follows we give more details on how the opponent end-point generates valid inputs, how the fuzzer handles encrypted messages, and why we use a dynamic memory analysis tool for failure detection.

A. Generating Inputs

We define the input to the SUT as the sequence of messages it receives in a single protocol execution. To generate an input in our test setup, depicted in Figure 1, we instruct the opponent end-point to initiate a new protocol session. The two end-points execute the protocol and the fuzzer gradually (i.e., message by message) obtains the input to the SUT, and may choose to mutate parts of the input. This process can be repeated to generate (and mutate) more valid inputs. The input generated in a protocol execution typically depends on the opponent end-point's configuration, which can be in the form of, e.g., configuration files, command line arguments, etc. Therefore, to generate a different input we restart the opponent end-point with a new configuration.

B. Fuzzing Encrypted Messages

Encrypted messages pose a challenge to the fuzzer because it cannot apply fine-grained mutations to them, for example to modify the value of a field in the message. In our test setup, the opponent end-point communicates the information necessary for decryption (i.e. encryption keys, random numbers used in encryption, and pointers to cryptographic algorithms) to the fuzzer using a log file. If the opponent end-point does not support such logging features, one may pick an open-source implementation of the security protocol under test, and add this functionality to the opponent of the SUT. Note that the opponent and the SUT need not be the same software (see above).

We remark that the fuzzer only learns the name of the cryptographic algorithms (e.g. AES) and the concrete instantiation of the algorithm (e.g. Cipher Block Chaining mode with 256 bits key length); hence the fuzzer must have access to an implementation of the cryptographic algorithms in order to perform encryption and decryption. Typically, security protocols use standard encryption algorithms, which allows the fuzzer to use an external cryptographic library. Note that the fuzzer decrypts a message only when mutating an encrypted message; other messages are directly passed to the SUT. This reduces the overhead of fuzz testing. Indeed, in our experiments with SecFuzz, the triangular routing through the fuzzer (as shown in Figure 1) is at most two times slower than the direct communication between the end-points.

C. Detecting Vulnerabilities

An integral part of any form of testing is the oracle that issues pass/fail verdicts. We use dynamic memory analysis tools that monitor the SUT's internal behavior as opposed to tools that monitor only external behaviors. For our case study we used Purify [14] and Valgrind [15].

The use of dynamic memory analysis tools benefits us in several ways. First, we gain precision as they detect vulnerabilities that may be missed by simply monitoring the I/O behavior of the SUT. For instance, reading a block of memory shortly after it has been freed often does not result in an obvious failure and its detection requires observing the system's internal behavior. Second, the aforementioned tools do not report false positives, which makes SecFuzz a sound tool. Finally, dynamic analysis tools typically provide detailed information about the system's state when a failure is detected; thus, locating the fault that caused the manifested failure is greatly simplified. A drawback of dynamic analysis tools is that they often incur a significant run-time overhead, which in some cases may slow down the SUT up to fifty times. In our experiments, this overhead is acceptable when monitoring end-points of security protocols, which are relatively small programs.

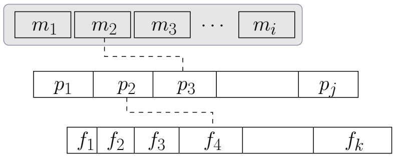

Figure 2. Structure of the input to security protocols. m, p, and f, denote a message, a payload, and a field, respectively.

In our case study, we have used dynamic analysis tools that focus on memory errors. Indeed, memory errors are the most common source of security vulnerabilities, such as denial of service, remote code execution, etc. However, dynamic analysis is in general not bound to memory error detection; one may employ dynamic analysis tools for other purposes such as dynamic invariant detection [16], e.g., for checking security invariants.

III. Fuzz Operators

Fuzz-testing is intrinsically incomplete: the input domain is often infinite and it is infeasible to test the system against all possible inputs. The effectiveness of fuzz-testing therefore depends on the fuzz operators' ability to produce "good" mutated inputs; that is, mutated inputs that expose *vulnerabilities* (i.e. faults that can be exploited by attackers) in the system.

In this section, we present a set of fuzz operators tailored for mutating inputs to security protocols. We first describe the inputs' structure. Then, we define our fuzz operators and explain their relevance for exposing real software vulnerabilities. We empirically demonstrate the effectiveness of our fuzz operators in a case study presented in Section IV.

A. Input Structure

We define the input to a security protocol's end-point as the sequence of messages it receives in a single execution of the protocol. Specifications typically define the internal structure of each message as a list of payloads. The structure of each payload is then defined as a list of fields, each

Table I
FUZZ OPERATORS FOR SECURITY PROTOCOLS

Category	*Operator's Name*	*Description*
Fuzz messages	Insert random message	Inserts a well-formed message at a random position in the messages sequence
Fuzz payloads	Insert random payload	Inserts a random payload at a random position in the list of payloads
	Duplicate random payload	Duplicates a randomly chosen payload
	Remove random payload	Removes a randomly chosen payload
Fuzz fields (numerical)	Set to random number	Sets the field to a randomly chosen number
	Set to zero	Sets the field to zero
Fuzz fields (strings)	Append random bytes	Appends a sequence of random bytes
	Modify random byte	Replaces a randomly chosen byte with a random byte
	Set to the empty string	Sets the field to the string of length zero
	Insert string termination	Inserts the string termination character at a randomly chosen position

field defining how the payload bits are interpreted. Figure 2 illustrates the input's structure.

In order to fuzz messages, the fuzzer needs to "understand" the structure of the input, i.e. to be able to distinguish different fields in a bit string that represents a message. This does not pose a challenge in practice: the message formats for a large number of security protocols have been specified, and the corresponding parsers and message constructor functions have been implemented in public libraries such as Scapy [17]. See Section IV-B for more details.

B. *Fuzz Operators*

The layered structure of the inputs to protocol implementations (see Figure 2) allows us to mutate them at different levels of abstraction. We categorize our fuzz operators into three classes, operators for fuzzing (1) messages, (2) payloads, and (3) fields. We further classify the field fuzz operators into operators for fuzzing numerical fields and fuzzing string fields, depending on whether a field holds a number or a string, respectively. Our fuzz operators for each of these classes are defined in Table I. Note that some operators are subsumed by others. For instance, *Set to zero* is a special case of *Set to random number*, and *Duplicate random payload* is subsumed by *Insert random payload*. We however define these special cases as separate operators, because testing boundary values has been proven particularly effective in exposing programming errors; see e.g. [18]. We remark that our fuzz operators are general and they can be applied to fuzz-testing different security protocols.

In what follows we present our fuzz operators in more detail. For each class of fuzz operators we state a hypothesis and argue that if the hypothesis holds, then our operators can expose vulnerabilities that are introduced by common programming mistakes. We justify our hypotheses by referring to common vulnerability databases.

1) Fuzzing fields and payloads: The field and payload fuzz operators mutate the internal structure of individual messages. The payload fuzz operators add, remove, or duplicate payloads. This results in missing or extra payloads in the message. The field fuzz operators mutate the values stored in the fields of a payload. Numerical fields often indicate the type or length of other fields/payloads. In contrast, string fields hold values that are typically used as inputs to the system's internal functions and hence they often have specific syntax, for example a string representing a date.

Note that a message often has dependencies across its fields. For example, a string field with variable length may have a dedicated field indicating the string's length; we say that the value of the length field depends on the string field. Similarly, each payload often has a field indicating the type of the next payload in the message. In our experiments we observed that the SUTs preprocess the received messages using packet filters, which often block messages with inconsistent fields. Therefore, to ensure that not all mutated messages are blocked, we update the dependent fields after applying a field/payload fuzz operator. This does however not imply that mutated inputs are always consistent. In the example above, the length field does not have dependent fields; hence mutating the length may result in an inconsistent input.

Hypothesis: Programmers often fail to properly validate inputs. Proper input validation refers to verifying the input's structure for compliance with the protocol's specification. This includes checking for valid input length, input type (e.g., "road" is a valid sequence of characters; however, it is invalid if the field must contain the name of a month), consistency across fields, missing and extra inputs, etc. Improper input validation may fail to reject an input with invalid structure, which can lead to undesired system behavior.

Argument: Our fields and payload fuzz operators mutate the input's structure. If the hypothesis holds, then these operators can expose failures.

Justification: According to the Common Weaknesses Enumeration database (CWE), improper input validation faults are prevalent and dangerous [1]. The CWE defines "improper enforcement of message or data structure" as a general class of software weaknesses such as "improper handling of syntactically invalid structure", "improper handling of unexpected data type", and others [19]. These software weaknesses directly relate to and support our hypothesis.

2) Fuzzing messages: The message fuzz operators mutate the sequence of messages sent to the SUT. We define the *Insert random message* operator for this purpose. This operator takes a well-formed message (e.g. captured from a previous session, or an older message in the current session of the protocol) and inserts this message at a random position in the message sequence. As a result, the SUT receives several valid messages followed by a message with valid structure, but at an unexpected position in the message sequence.

Hypothesis: Programmers often fail to properly bridge the abstraction gap between the specification and the implementation. While the specification is often input-incomplete, i.e. it does not specify the system's behavior for all possible inputs, the implementation must be prepared to handle any input. Programmers often fail to correctly map the abstract specification to an input-complete implementation. This may result in inadequate exception handling, which in turn can introduce vulnerabilities.

Argument: If the hypothesis holds, the message fuzz operators can expose failures.

Justification: Transforming an abstract specification into a concrete system is well known to be a difficult problem. In Section IV we demonstrate a previously unknown vulnerability that was exposed by feeding the SUT with an unexpected message, which further supports our hypothesis.

We remark that the effectiveness of the fuzz operators presented in this section relies on the aforementioned hypotheses. In addition to the case study presented in Section IV, we intend to test the validity of these hypotheses by applying SecFuzz to a variety of security protocols in the future.

IV. Case Study on the Internet Key Exchange Protocol

To evaluate the effectiveness of our fuzz-testing technique we conducted a case study on the Internet Key Exchange (IKE) protocol [20]. IKE is used for key exchange within IPSec, which is commonly used to construct virtual private networks. We chose two mature, widely used IKE implementations as our test subjects: OpenSwan [21] and Shrew Soft's VPN Client [22].

We start with a brief overview of IKE. Then we present our tool for fuzz-testing security protocols, called SecFuzz. Next, we describe our experiments on the aforementioned IKE implementations. Finally, in the context of our case study we compare SecFuzz to two existing model-based fuzz-testing tools for IKE, namely Protos and IkeFuzz.

A. The Internet Key Exchange Protocol

IKE is used to set up security associations between two end-points. A security association (SA) is a set of cryptographic attributes (e.g. keys, cryptographic algorithm, etc.) and a security policy used to protect information. IKE uses the Internet Security Association and Key Management Protocol (ISAKMP), which is a framework for authentication and key exchange [23]. The protocol proceeds in two phases. The first phase sets up an ISAKMP SA, which is used to establish a secure channel for further communication between the end-points. The purpose of the second phase is to set up a security association on behalf of another service, such as IPSec.

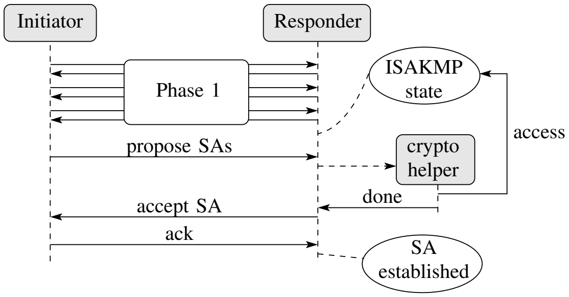

Figure 3. An abstract view of Openswan's implementation of IKE. A data race can cause the crypto helper thread to access freed memory.

B. The SecFuzz Tool

The SecFuzz tool supports the fuzz operators defined in Section III and relies on Scapy, a Python library for parsing and manipulating messages [17]. Currently, the implementation of SecFuzz is limited to IKE messages; nevertheless, it can be easily extended to other message formats (e.g. IEEE 802.11 authentication protocol) due to Scapy's flexibility.

SecFuzz executes as follows. Each protocol execution between the SUT and its opponent end-point constitutes one test case. SecFuzz listens for messages destined for the SUT and applies one fuzz operator per test case. To detect the beginning of a new test case, SecFuzz checks if a message belongs to a new protocol session. For each new session SecFuzz retrieves the corresponding cryptographic attributes (e.g. key, cryptographic algorithm, etc.) from the opponent end-point's log file. When a new test case begins, SecFuzz chooses uniformly at random a fuzz operator and a position at which the input is mutated. The tool counts the messages sent to the SUT and applies the fuzz operator when the position for fuzzing is reached. For field and payload fuzz operators the position for fuzzing indicates which message is mutated and for the message fuzz operator it indicates the place where the message sequence is modified.

C. Experiment 1: OpenSwan

In this section we present our experiment on OpenSwan, which is an open-source IPSec implementation for Linux. It is available on most popular Linux distributions, including RedHat and Debian. We first describe how we instantiate our SecFuzz test setup and then we report on a previously unknown vulnerability discovered by SecFuzz.

1) The Setup: The SUT in this experiment is the IPSec's responder in OpenSwan version 2.6.35. This was the latest version at the time of conducting our experiment. The SUT is executed inside Memcheck [24], which is a memory error detector for C and C++ binary programs based on Valgrind [15]. It can detect a wide range of memory errors such as use of undefined variables, invalid memory access, incorrect heap memory management, memory leaks, and others.

The opponent end-point to the SUT is OpenSwan's initiator. To ensure that the opponent end-point generates different inputs, we automatically generated OpenSwan configuration files and every 60 seconds the opponent is restarted with a new configuration file. When configured in debug mode, OpenSwan logs the cryptographic keys, algorithms, etc., to a file. To allow SECFUZZ to decrypt messages, we set the debug flag in OpenSwan's configurations and provided the fuzzer with remote access to this log file.

We execute the opponent end-point, the SUT, and the fuzzer on different virtual machines connected to the same local network. To force the messages destined for the SUT to pass through the fuzzer, we modify the ARP table of the opponent's machine by mapping the SUT's IP to the fuzzer's MAC address.

2) The Results: We found a use-after-free memory access problem. Use-after-free vulnerabilities are serious as they can, with a high likelihood, be exploited [19]. As pointed out in Section II-C, dynamic memory analysis tools can greatly simplify the process of locating faults. Indeed, Memcheck reported the stack traces of the threads that accessed and freed the memory. This detailed information was sufficient for us to inspect the code and pinpoint the problem's cause:

Figure 3 shows an abstract view of OpenSwan's IKE implementation. OpenSwan stores the ISAKMP SA's state information established after IKE's phase 1 in a specific data structure (depicted as "ISAKMP state" in Figure 3). In phase 2, the initiator proposes a set of SAs to the responder. The responder selects an SA and computes the necessary cryptographic attributes for the new SA (e.g. key, initialization vector, etc.). OpenSwan creates a "crypto helper" thread for this purpose, which keeps a pointer to the ISAKMP state. Then, the responder replies with an "accepted SA" message. Finally, the initiator sends an acknowledgment to demonstrate its liveness and readiness to use the new SA. [1]

The vulnerability is exposed when the data structure storing the ISAKMP state is freed before the crypto helper thread accesses the state. This may happen when the initiator sends a "close ISAKMP session" message immediately after the "propose SAs" message. After receiving this message, the responder closes the ISAKMP session and frees the memory storing the ISAKMP state. This introduces a data race: if the memory is freed before the crypto helper accesses it, then the thread accesses freed memory. This vulnerability is exposed by our *Insert random message* fuzz operator.

Table II
AVERAGE TIME TO REVEAL VULNERABILITIES BY SECFUZZ, PROTOS, AND IKEFUZZ

Vulnerability \ Fuzzer	SECFUZZ	PROTOS	IKEFUZZ
1. CVE-2011-3380	✗	✓ 94 min.	✗
2. CVE-2011-4073	✓ 80 min.	✗	✗
3. CVE-2012-0783	✓ 10 min.	✗	✗
4. Vulnerability 4	✗	✓ 94 min.	✗

We reported the vulnerability to OpenSwan's developers, who then released a security patch. More details on the vulnerability and the patch can be found in CVE-2011-4073.

D. Experiment 2: Shrew Soft's VPN Client for Windows

In the second experiment we tested Shrew Soft's VPN Client, which is a popular IKE implementation for Windows.

1) The Setup: For dynamic analysis we used IBM's Rational Purify [14]. This tool detects memory access errors (e.g. uninitialized memory access, buffer overflows, improper memory frees) and memory leaks. The SUT is Shrew Soft's VPN Client, which implements IKE's initiator. For the experiment we used version 2.1.7 of the software, which again was the latest version at the time of conducting the experiment. The opponent end-point to the VPN Client is OpenSwan's responder. The rest of the experimental setup is similar to the one used in our OpenSwan experiment.

2) The Results: We found an unhandled exception vulnerability in the VPN Client. The source code of the implementation is not available and we could not investigate the error in detail. We reported the vulnerability to the developers, who confirmed and fixed the vulnerability. The vulnerability is exposed by the *Set to the empty string* fuzz operator and occurs before authentication; hence it can be exploited by any attacker. We expect a patched version of the Shrew Soft's VPN Client to be released before the publication of CVE-2012-0783, which will detail the vulnerability.

E. SECFUZZ *Compared to Other IKE Fuzzers*

In this section, we report on the effectiveness of SECFUZZ in terms of the "number of found failures" and the "average time to find a failure". [2] These metrics can be used by software developers and managers to perform risk analysis and decide whether they should fuzz-test their product. Moreover, we use these metrics to compare the effectiveness of SECFUZZ to two model-based fuzzers tailored for IKE implementations, namely PROTOS [4] and IKEFUZZ [5]. We remark that these model-based fuzzers are significantly more complex than SECFUZZ (e.g. IKEFUZZ has 16901 lines of

[1] These are the three messages exchanged in IKE Quick mode [20].

[2] We use *average* time to find failures because fuzz-testing is inherently probabilistic.

code versus 876 in SecFuzz) as they use a model of the IKE protocol to generate valid inputs.

For our comparisons, we used OpenSwan version 2.6.35 and Shrew Soft's VPN Client version 2.1.7 as test subjects. We ran the experiments on Linux virtual machines using a Thinkpad T420 laptop and provided each fuzzer with 10 hours for testing each IKE implementation. We then measured the average time the fuzzers needed for each vulnerability they found. Table II summarizes our results.

As Table II shows, Protos and SecFuzz each found two vulnerabilities in the test subjects; in our experiments IkeFuzz could not discover any failures. Interestingly, the vulnerabilities found by Protos and SecFuzz are different. In what follows, we further explain the results of Table II.

The CVE-2011-3380 vulnerability is exposed when OpenSwan receives an SA payload with an invalid key length. The key length is specified in a field holding a list of SA attributes. SecFuzz currently cannot mutate fields that store lists of values; hence SecFuzz could not mutate the key length attribute in order to expose the vulnerability.

The second vulnerability is described in Section IV-C. The problem is exposed when OpenSwan receives an unexpected "close session" message in IKE's phase 2. Protos is limited to mutating only phase 1 messages of IKE, and moreover it does not have an equivalent to our *Insert random message* fuzz operator. These limitations prevent Protos from discovering this vulnerability.

The third vulnerability pertains to Shrew Soft's VPN Client. Protos and IkeFuzz are limited to fuzz-testing IKE responders, and the VPN Client implements IKE's initiator end-point. Thus, only SecFuzz can fuzz-test Shrew Soft's VPN Client.

Vulnerability 4 pertains to the use of IP6 in OpenSwan. In our experiments with SecFuzz we decided to confine our fuzz-testing to IP4, and therefore did not generate IP6 configurations. SecFuzz could in principle be configured for IP6, and would then discover this vulnerability (see Section II). We have communicated this (previously unknown) vulnerability to the OpenSwan development team.

To summarize, SecFuzz appears as effective as Protos, and they both are more effective than IkeFuzz. SecFuzz has several major advantages over Protos: it can fuzz-test any IKE implementations (i.e. initiator and responder end-points), it can mutate phase 2 messages, and it can modify message sequences by inserting well-formed messages. SecFuzz is weaker than Protos in two regards: SecFuzz currently cannot mutate fields storing lists of values, and also it depends on the opponent end-point's configurations. These limitations are however straightforward to overcome.

Acknowledgments

The work has been partially supported by the EU FP7 projects Spacios (no. 257876). We thank Paul Wouters (OpenSwan) and Matthew Grooms (Shrew Soft) for assisting us with investigating the two vulnerabilities. We also thank Hubert Ritzdorf for his help with the IKE case study.

References

[1] T. M. Corporation, "2011 CWE/SANS Top 25 most dangerous software errors," Sep. 2011. [Online]. Available: http://cwe.mitre.org/top25/index.html

[2] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services," *Office*, vol. 1525, no. October 1995, pp. 1–23, 1995.

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, pp. 32–44, December 1990.

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

[5] "IkeFuzz," https://www.ee.oulu.fi/research/ouspg/ikefuzz.

[6] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[7] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[8] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 209–224, 2008.

[9] 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.

[10] T. Alrahem, A. Chen, N. DiGiuseppe, J. Gee, S.-P. Hsiao, S. Mattox, T. Park, I. G. Harris, and et al., "Interstate: A stateful protocol fuzzer for SIP," in *Defcon 15*, August 2007, pp. 1–5.

[11] R. Kaksonen, M. Laakso, and A. Takanen, "System security assessment through specification mutations and fault injection," in *Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues of the New Century*. Deventer, The Netherlands: Kluwer, B.V., 2001, pp. 27–.

[12] Y. Hsu, G. Shu, and D. Lee, "A model-based approach to security flaw detection of network protocol implementations," *2008 IEEE Intl. Conf. on Network Protocols*, pp. 114–123, 2008.

[13] Codenomicon, "The Defensics traffic capture fuzzer." [Online]. Available: http://www.codenomicon.com

[14] R. Hastings and B. Joyce, *Purify: A Tool for Detecting Memory Leaks and Access Errors in C and C++ Programs*. USENIX, 1992, pp. 125–138.

[15] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[16] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the 22nd international conference on Software engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 449–458.

[17] "Scapy," http://www.secdev.org/projects/scapy/.

[18] C. Wysopal, L. Nelson, E. Dustin, and D. Zovi, *The Art of Software Security Testing: Identifying Software Security Flaws*. Pearson Education, 2006.

[19] T. M. Corporation, "CWE-List (2.1)," Sep. 2011. [Online]. Available: http://cwe.mitre.org/data/

[20] D. Harkins and D. Carrel, "The Internet Key Exchange (IKE)," RFC 2409 (Proposed Standard), Internet Engineering Task Force, Nov. 1998, obsoleted by RFC 4306, updated by RFC 4109. [Online]. Available: http://www.ietf.org/rfc/rfc2409.txt

[21] "Openswan," https://www.openswan.org.

[22] "Shrew Soft VPN Client," http://www.shrew.net.

[23] D. Maughan, M. Schertler, M. Schneider, and J. Turner, "Internet Security Association and Key Management Protocol (ISAKMP)," RFC 2408 (Proposed Standard), Internet Engineering Task Force, Nov. 1998, obsoleted by RFC 4306. [Online]. Available: http://www.ietf.org/rfc/rfc2408.txt

[24] "Memcheck: A memory error detector," http://valgrind.org.