Extension of SPIKE for Encrypted Protocol Fuzzing

Aabha Biyani
Gantavya Sharma
Jagannath Aghav
Piyush Waradpande
Purva Savaji

Department of Computer Engineering
College of Engineering
Pune, India
espike.coep@gmail.com

Mritunjay Gautam
Symantec pvt. ltd.
Pune, India
mrityunjay@gmail.com

***Abstract*—A fuzzer is a program that attempts to find security vulnerabilities in an application by sending random or semi-random input. Fuzzers have been widely used to find vulnerabilities in protocol implementations. The implementations may conform to the design of the protocol, but most of the times some glitches might remain. As a result vulnerabilities might remain unnoticed. Consequently, different implementations of the same protocol may be vulnerable to different kind of attacks. Fuzzers help us discover such implementation flaws. Among the currently available and popular ones, SPIKE is one recognized open-source fuzzing framework. However, SPIKE has a limitation of fuzzing only non-encrypted protocols. This paper presents the extension of SPIKE, called ESPIKE, for fuzzing of encrypted protocols. ESPIKE will facilitate testing implementations of SSL encrypted protocols. As a proof of concept for efficiency of ESPIKE we demonstrate its usage on sftp and https protocol.**

Keywords– Fuzzing, Encrypted protocols, Security, Testing

I. Introduction

Fuzzing is a process of intentionally sending malformed data to a program under test in order to discover and analyze its failures. It involves basic steps of structuring data to send, sending this data, analysing target behavior and discovering bugs. Traditional testing methodologies focus on testing completeness or correctness. Instead, Fuzzing complements traditional testing to find out security loop holes in the form of untested combinations of code and data by combining the power of randomness and protocol knowledge[1]. As a result fuzz testing forms important part of security testing. The first (or at least best known) rudimentary negative testing project was called Fuzz from Barton Miller's research group at the University of Wisconsin, published in 1990[6]. Basic tests involved in the project were all command line fuzzing attacks.

Automated fuzzing tools like SPIKE[8], Peach[10], beSTORM[11], MU-Dynamics, etc. are available to perform fuzz testing. SPIKE is an open-source fuzzer creation tool-kit. It is an API written in C which contains all the functions required to write a fuzzer. Its large database of fuzz strings which provides an intelligent input set to find the software faults such as buffer overflow, integer overflow and memory allocation bugs.

Nowadays for obvious security reasons, encrypted protocols are used for communication. Due to the use of SSL the data over network is not in recognizable format. So encrypted protocols like https, sftp etc are widely used over network. But there are flaws in these encrypted protocol implementations too. For discovering these flaws encrypted protocol fuzzers are needed. There are commercially available tools for that purpose. Although, SPIKE lacks this functionality of fuzzing encrypted protocols. Hence, need arises to make SPIKE API more powerful by adding the functionality of fuzzing encrypted protocols. With this motivation, we extended SPIKE to ESPIKE ie. Encrypted-SPIKE which is capable for encrypted protocol fuzzing. This paper explains the design and implementation of ESPIKE and the proof of concept for ESPIKE.

II. Literature Review

Fuzzing works on the principle of the Monte Carlo method[13] to identify implementation flaws. Monte Carlo methods rely on repeated random sampling to compute results. Fuzzing too repeatedly generates random inputs to obtain results of identified vulnerabilities. Fuzzing has a capacity of finding implementation level flaws. However, it cannot find design problems. For example, buffer overflow can be detected but use of weak encryption algorithm cannot be detected by using fuzzing.

With increased popularity of fuzzers, even end-users or clients employ complex fuzzing methods before deploying the software in their own company. This shows how important it is to have good quality fuzzers that are customizable and adaptable to all kinds of protocols and applications. There are professional grade fuzzers like Codenomicon[7], SPIKE, Sulley[9], Peach, beSTORM, MU-Dynamics etc. available in the market. They use complex algorithms and

efficient input sets that are designed to find out bugs and vulnerabilities. A thorough comparison[14] between the available fuzzers will highlight their merits, drawbacks and suitability for the purpose being served. Customizable fuzzers available allow the user to specify fuzzing parameters and also provide in-built database of attack strings. Usually such generic fuzzers come with attack strings that are most likely to highlight faults. At the same time the ability to customize make them more flexible.

SPIKE is a block based fuzzer. It has simple scripting capability, allowing interpretation of simple text files containing SPIKE primitives. Being block based, it helps to find vulnerablilties in various protocol implementations with ease[2]. SPIKE, currently is able to fuzz or test only non-encrypted protocols. Encrypted protocols like https, sftp etc are not accounted for. In real world scenarios, due to confidentiality reasons, secured protocols are widely used and prefered. OpenSSL based encryption is widely used to secure the communication taking place on the wire. The protocols use end-to-end application-level encryption which makes it difficult to understand whether they can withstand injection of bad data. This situation makes it hard to apply fuzzing techniques to test their security and reliability. Due to lack of this ability in SPIKE, need arises to extend this fuzzer to work for encrypted protocols. However, fuzzing of encrypted protocols could be a bit cryptic. Most fuzzers operate as intermediary or proxy applications and would need to either decrypt communications on the fly or have the client program's cooperation in muxing the traffic. We intend to build an extension to SPIKE that will implement SSL based communication between SPIKE and the SSL based target server. By doing this we will eventually be able to increase the scope of SPIKE fuzzer by almost two folds. Some security scanners and fuzzers provide such capability. One such security scanner tool, skipfish[12], allows fuzz-testing over https, but like most of the others, it is intended only towards web-based scanning. SPIKE on the other hand is very generic and can be applied to almost any protocol. Thus our extended version of SPIKE i.e. ESPIKE would be more generic allowing capability to fuzz encrypted versions of wide raange protocols already compatible with SPIKE.

III. Proposed Solution

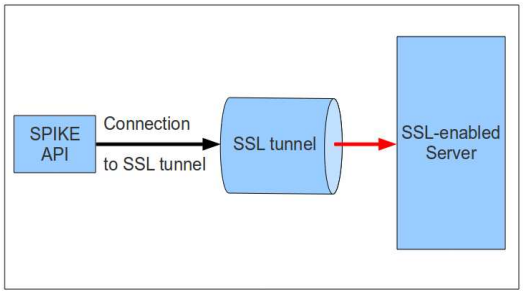

Fig. 1. Tunnel view of ESPIKE

For creating ESPIKE, a SSL wrapper is to be provided around the SPIKE API. This wrapper will handle the SSL communication. The SPIKEs generated should be sent to the SSL-enabled target via the SSL wrapper. This wrapper around SPIKE can be visualised as a tunnel between SPIKE and SSL-enabled target. The following figure represents the tunnel view of ESPIKE.

The fuzz strings generated by SPIKE are given as input to the SSL tunnel. We thus need to provide a communication channel between the Fuzzer Program and the SSL Tunnel. This communication channel could be shared memory, message based system like pipe[15] or message queuing. Message queueing in this case would be a time consuming affair. It would also require extra memory and lot of extra CPU cycles. Shared memory suffers from two main disadvantages, protection and synchronization. Hence, pipe would be an efficient option for communication. To use pipe as a communication channel, one of the process should be forked and executed from the another. Once the pipe is created it elegantly handles all the data from the fuzzer program and sends it to the SSL tunnel which connects the SPIKE fuzzer to the server. Use of pipe scores in speed and efficiency too.

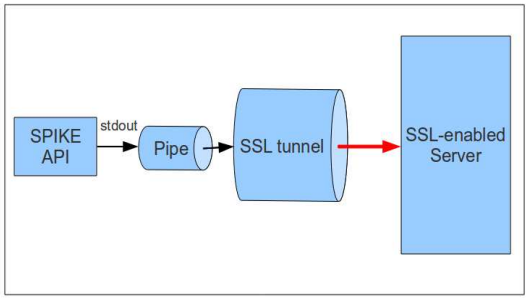

Fig. 2. Integration using Pipe

IV. ESPIKE

A. Design

1) SSL Object Initialiser: This module forms the core of ESPIKE. It creates the SSL layer that takes care of all the encrypted communication. Maintaining the SSL session throughout the fuzzing process is essentially the responsibility of this module. It includes library and error initializations, managing the certifying authorities and private key for ESPIKE, cross-referencing the root CA list for being assured of security, maintaining the SSL context throughout the session and other such tasks. Standard OpenSSL programming has been used to design this module. Once this module establishes a secure communication has to be linked with the SPIKE API. The subsidary modules help us achieve this.

2) TCP Socket Initialiser: TCP Socket initialiser is used to create a socket and returns a socket id. It thus establishes a connection at Network Layer. The socket id returned is further used for establishing SSL connection.

3) SSL Connector: A SSL connection object is created in this module. This object provides the connection at the transport layer where the SSL protocol is implemented. We need to attach the socket at network layer to the ssl object at the transport layer. Hence, BIO object is created using socket, which is later attached to the ssl object.

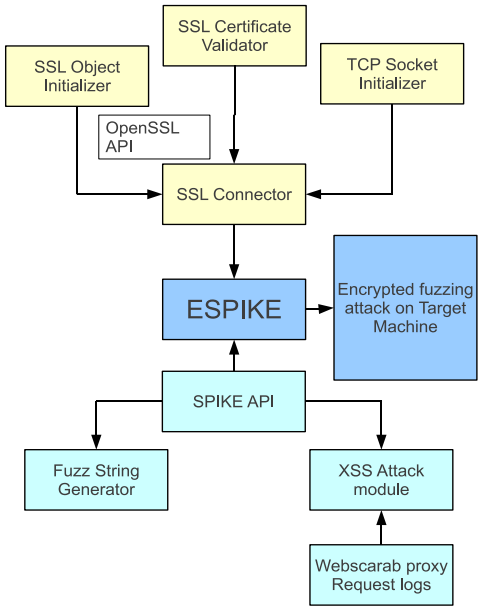

Fig. 3. Architecture of ESPIKE

4) Fuzz String Generator: Fuzz strings are generated using the SPIKE API. User has to take the precaution of using s_printf_buffer() instead of spike_send() function in the SPIKE program that generates the fuzz strings. Instruction regarding this can be found in the next section.

B. Porting SPIKE to ESPIKE

With the addition of the SSL layer, it was required to make minor changes in the SPIKE files. In this section we discuss those changes and document the instructions needed to port existing SPIKE programs to ESPIKE. To understand in detail the changes, the reader should refer the README_ESPIKE.txt file bundled along with ESPIKE. The changes documented below are with reference to SPIKE 2.9.

1) spike.c:
   a) In the s_push() function, on line number 167, as a parameter to realloc() the size has been changed to size+1

```
SPIKE:tmp=realloc(current_spike->
databuf, current_spike->datasize+size);

ESPIKE:tmp=realloc(current_spike-
> databuf, current_spike->datasize
+size+1);
```

   b) On line number 175, in memcpy function the last argument has also been changed to size+1

```
SPIKE: memcpy(current_spike->
databuf+current_spike->datasize,
pushme, size);

ESPIKE: memcpy(current_spike->
databuf+current_spike->datasize,
pushme, size+1);
```

   c) The printf statement in s_string_variable() function, line number 1872, and that in s_init_fuzzing() function, line number 2388 are commented

```
SPIKE: printf("Variablesize= %d",
size2-size);

ESPIKE: //printf("Variablesize= %d",
size2-size);

SPIKE: printf("Total Number of Strings
is %d", maxfuzzstring);

ESPIKE: //printf("Total Number of
Strings is %d", maxfuzzstring);
```

2) Makefile: New targets have been added to compile the newly added files. The targets added are convert_attacks, fuzz_ftp_sample, generic_ssl_client, xss_ssl_client, xss_attack_spike.

As pipe is used to connect SPIKE to the SSL tunnel, the user should take care of the data that is being printed on the stdout. pipe() transfers that content completely to the SSL tunnel. Hence any excess data printed on the stdout will be treated as false input to the SSL module. For this purpose, the user is required to print the contents of data buffer in SPIKE where the fuzz strings are stored. s_printf_buffer() function will suffice for this purpose. All other printf() statements should be eliminated from the program. Also the tcp connection is established by the SSL layer, requiring the user to eliminate connection related calls like spike_tcp_connect(), spike_tcp_close(), spike_send() and spike_tcp_send().

V. Experimentation

Finally after the successful integration of OpenSSL and SPIKE API, to test its correct functioning, we have built two POCs. One was to test by fuzzing a sftp server and the other one being a module to test XSS attacks over HTTPS.

A. Filezilla

To test out a sftp server we tried it out on FileZilla Server running under Windows environment. Configuring

a sftp server on FileZilla is fairly straight-forward, requiring some very basic SSL housekeeping. All you need to provide is a private key and a certificate. Once the sftp server was set up and functioning, we deployed the ESPIKE module to carry out its field-test. We tried to fuzz the sftp server using the standard ftp commands like USER, PASS etc. Fuzz testing on the sftp server was successfully carried out with expected error and success responses from the server showing its ability to communicate under the SSL cover. Same set of fuzz-string inputs were supplied using SPIKE to the plain ftp server to verify the correctness of the newly built tool. Exactly same result when fuzzing with the non-encrypted and encrypted variant of the same ftp software ie. Filezilla indicate the correct functioning of ESPIKE.

The command below demonstrates the usage of the ESPIKE for fuzzing a sftp server.

$ ./generic_ssl_client ServerHost ServerPort SPIKE_fuzzer_program

Options provided in the above command are fairly comprehensive. ServerHost being the IP address of the sftp server, ServerPort being being the port on which it is running and finally SPIKE_fuzzer_program is the binary of the modified SPIKE program written to fuzz the normal ftp server. Guidelines on converting existing SPIKE programs to be compatible with ESPIKE are mentioned in the section 4 B. For detailed guidelines the user is recommended to go through the documentation provided with the ESPIKE source.

B. XSS

In order to provide a concrete PoC (Proof of Concept), we have designed the XSS attack module. XSS or Cross Site Scripting attacks are commonly carried out by security experts to verify and correct faults in their web-applications, or by malicious attackers to discover and exploit vulnerabilities. It is thus a good idea for a developer to test his application for such attacks and this module provides with such functionality. XSS is carried out with an intention that the input entered by the user is treated as a part of the code instead of it being treated as data. For instance, a user could enter a javascript like `<script>alert('xss');</script>`[5] where the "Name" of user is expected. Due to lack of input validation, this script gets accepted as any other normal input, however it gets executed when it is fetched from the user. XSS attack module thus demonstrates a real-world application of the ESPIKE which could be used by security professionals for securing their web-applications. Tools such as Skipfish are great security scanning tools, but are not directly intended towards XSS detection. They have a wider domain hence their design is considerate of cost-benefit calculations. Whereas ESPIKE XSS module is designed with XSS attacks as prime focus. Currently only GET requests are supported, but a small tweak in the code will include POST requests as well. The work is in progress. Another interesting factor with this module is that it allows the user to selectively fuzz different parameters so that the fuzzing process is more targeted, focused and hence result-oriented.

1) XSS module components: XSS module has five main components. We discuss the functionality of each in detail.

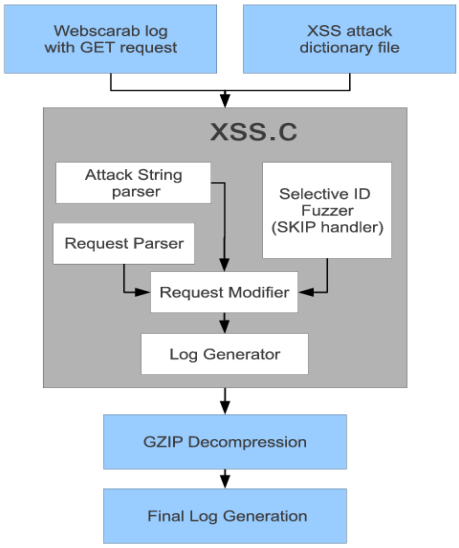

Fig. 4. XSS Module

1) Attack parser:
   This component stores the attack strings into an array that are used as fuzz strings for id values of the GET request method. It does so by parsing a dictionary file where all the attacks are stored. During the parsing operation, the default values are stored in appropriate structure to facilitate selective fuzzing or fuzzing of individual elements. As a subcomponent, the convert_attacks.c converts all attacks from hex-encoded format to ASCII format for easy handling.
2) Log parser:
   The log parser parses a request log that will be used as a template for fuzzing. We used Webscarab[3] logs as it provided neat logs without any extra information. The log parser separates the host information and the GET request parameters along with their default values. The default values are separately stored in specific structures as the user might be interested in selective fuzzing.
3) Request modifier:
   This is a part of the xss_espike.c program that parses the various fields of the request from the request-logs. It separates the part where the attack strings are to be injected, various ids and their values. These separated ids and values are used by the request modifier to carefully modify the requests that are to be sent as fuzz strings. Request modifier collaborates

with the SKIP handler to selectively fuzz only the intended ids as specified on the command line.

4) Log generator:
Every response should have a proper log that can be understood by the user. This functionality is provided by Log Generator module of the program. Responses can be received in gzip format. Such responses are identified by the .gz extension. Also logs are separated according to attack strings. An attack number is prefixed at the beginning of the attack to identify which attack generates which response. All these logs are stored in a new directory created under the main SPIKE directory by name 'outputlogs'. A specific nomenclature is followed while naming the logs to facilitate easy review of logs. The logs are prefixed by the request numbers, followed by the keyword 'request', then followed by the logfile name. Sample logfile names are give below.
1-request.localhost_logfile
1-request.localhost_logfile.gz

VI. Conclusion

As encrypted protocols improve the security in network, the number of attacks against encrypted protocol implementation are also on the rise. Thus more and more powerful and intutive tools need to be developed to assess the security of network programs.

Fuzzing is evolving as a widely used method for testing which explores the loopholes that otherwise remained undetected. Hence, the idea of ESPIKE i.e. extending SPIKE for fuzzing encrypted protocols emerged.

ESPIKE sends all the SPIKE data through SSL layer so as to attack the encrypted protocol implementation. For that a SSL client program is developed using the fuctions of OpenSSL library which takes care of the initial handshakes and verification with the targeted server. The fuzz strings are generated using SPIKE fuzzer program. After a lot of debate and comparisons we have chosen the best suited approach of pipe for integrating the SSL client and SPIKE fuzzer. Pipe has proven to be efficient for memory and time. Minor modifications have been made to the existing SPIKE code, so that user can conveniently port from existing SPIKE programs to ESPIKE.

ESPIKE has been tested using Filezilla sftp server and xss module. The result of ESPIKE is similar to that of SPIKE. This confirms the successful working of ESPIKE. The xss module also adds to the examples of SPIKE API with the attacks dictionary. More such test modules can be written.

A user generally implements a lot of verbose information on the standard output. This is a very natural and widely used practice. But while modifying the existing SPIKE programs we need to redirect the verbose information from the standard output. It may be stored to a file as a log that can be reviewed later. The popen() system call sends all the data printed on stdout directly to the other end of the pipe where the SSL communication program resides. Any additional printing of characters will no doubt give rise to unexpected errors. So, in the correct use scenario, nothing should be printed on the standard output. For convenience of the user we have also bundled a README.ESPIKE file that clearly lays down the guidelines for converting and porting existing codes to ESPIKE.

The inclusion of this module in SPIKE will be helpful to security professionals and white-hat hackers to a great extent. The added functionality, is already present in commercial tools, but that is where the OpenSource community lacked. Through this project we tried to fill the gap and give back to the community. It will immensely help developers and security-tester working on free and open source products to carry out their work efficiently without use of expensive and commercial fuzzers.

VII. Acknowledgement

This project began as a B.Tech project offered by Symantec Inc. We are sincerely thankful to the company for providing us all with the technical expertise required for its successful completion. Mr. Mrityunjay Gautam, our mentor, deserves a special thanks for his very valued guidance to us throughout the project. The project would not have been possible without his support. Prof. J. V. Aghav, our professor from College of Engineering, Pune, was a constant source of motivation for us and helped us with every aspect of the project. Finally, we would also like to thank Dave Aitel for providing the community with the extremely useful SPIKE fuzzer.

References

[1] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer and Giovanni Vigna *SNOOZE: toward a Stateful Network protocol fuzzer,* Department of Computer Science University of California, Santa Barbara
[2] Dave Aitel, *The Advantages of Block-Based Protocol Analysis for Security Testing,* Immunity,Inc., February 4, 2002
[3] OWASP WebScarab Project- http://www.owasp.org/index.php/OWASP_WebScarab_Project/
[4] The OpenSSL Project- http://www.openssl.org
[5] XSS Cheat sheet- http://ha.ckers.org/xss.html
[6] http://pages.cs.wisc.edu/bart/fuzz/fuzz.html
[7] Codenomicon Webpage- http://www.codenomicon.com/
[8] SPIKE Webpage- http://www.immunitysec.com/resources-freesoftware.shtml
[9] Sulley Webpage- http://code.google.com/p/sulley/
[10] Peach Webpage- http://peachfuzzer.com/
[11] beSTORM Webpage- http://www.beyondsecurity.com/bestorm_whitepaper.html.
[12] Skipfish Webpage- http://www.code.google.com/p/skipfish/
[13] http://en.wikipedia.org/wiki/Monte_Carlo_method
[14] Ari Takanen, Jared DeMott, Charles Miller *Fuzzing for software security testing and quality assurance*
[15] Richard Stevens *Advanced Programming in the UNIX Environment, ch 18 "Pipes and Processes", Pgs-347-355*, Addison-Wesley, 1992, ISBN 0-201-56317-7.