Building a Graphical Fuzzing Framework

Sascha Zeisberger
Dept. Computer Science
Rhodes University
Grahamstown, South Africa
Email: g07z3446@campus.ru.ac.za

Barry Irwin
Dept. Computer Science
Rhodes University
Grahamstown, South Africa
Email: b.irwin@ru.ac.za

***Abstract*—Fuzz testing is a robustness testing technique that sends malformed data to an application's input. This is to test an application's behaviour when presented with input beyond its specification. The main difference between traditional testing techniques and fuzz testing is that in most traditional techniques an application is tested according to a specification and rated on how well the application conforms to that specification. Fuzz testing tests beyond the scope of a specification by intelligently generating values that may be interpreted by an application in an unintended manner. The use of fuzz testing has been more prevalent in academic and security communities despite showing success in production environments.**

To measure the effectiveness of fuzz testing, an experiment was conducted where several publicly available applications were fuzzed. In some instances, fuzz testing was able to force an application into an invalid state and it was concluded that fuzz testing is a relevant testing technique that could assist in developing more robust applications. This success prompted a further investigation into fuzz testing in order to compile a list of requirements that makes an effective fuzzer.

The aforementioned investigation assisted in the design of a fuzz testing framework, the goal of which is to make the process more accessible to users outside of an academic and security environment. Design methodologies and justifications of said framework are discussed, focusing on the graphical user interface components as this aspect of the framework is used to increase the usability of the framework.

***Index Terms*—Fuzzing, Application Testing, Frameworks**

I. Introduction

FUZZ testing, colloquially know as fuzzing, is a robustness testing technique that sends malformed data to an application's input [1]. It focuses on generating intelligent values that extend beyond the scope of an application for which developers may not have accommodated for and possibly sending an application into an unknown state [2]. These unknown states commonly manifest as the application crashing, and present a security risk as these crashes could lead to the application becoming exploitable to some extent [3]. Popular targets for undergoing fuzzing include application arguments and files and network communications. It is accepted that if an application accepts some form of input, that input can be fuzzed. As an example, Mulliner and Miller demonstrate that by fuzzing the SMS stack in several cellular phone operating systems, they were able to find issues in the respective platforms and were able to develop exploits in several instances [3].

Fuzz testing has been shown to be effective in development environments, more so when combined with other testing tools [4]. Despite this fact, fuzzing has not become widely used in commercial development environments and remains fairly restricted to academic and security backgrounds. This paper looks at the development of a fuzzing framework that aims to simplify the fuzzing process in order to increase its use in industry. Focus is put on fuzzing network applications at the application layer, specifically targeting FTP and SIP. This is to demonstrate that fuzz testing can be used in this format to reveal flaws in network applications, explain the process of fuzz testing in terms of the requirements before, during and after a fuzz session, and explain the benefits and shortcomings of fuzzing. This data is used to facilitate the design of the customised fuzzing framework whose goal is to simplify the process of fuzz testing and lower the learning barrier to conduct fuzz tests. Section II of this paper provides a short summary of the different aspects of fuzzing. Various concepts that relate to fuzzers specifically are discussed, focusing on the types of fuzzers, strategies of fuzzing and data generation methods. Section III describes the process of fuzzing two network protocols, providing summaries on the protocols fuzzed, the fuzzers used in each scenario and a short summary on the results from each fuzzing session. Section IV uses the discussions from the previous two sections to propose a new fuzzing framework and motivate any design decisions. Section V outlines several shortcomings of the developed framework and proposes several requirements that may improve the framework. Section VI concludes the discussion and re-iterates the more important topics covered in this paper.

II. Fuzzing

Fuzz testing aims to reveal implementation flaws in an application by sending malformed queries to the target application. Most developers measure an application's completeness according to its ability to conform to a specification, usually checking if an application accepts a set of correctly formatted inputs and responds in the expected manner [2]. Fuzz testing aims to test an application beyond this scope by generating intelligent values, where an intelligent value is one that could possibly be interpreted incorrectly by an application [1]. Intelligent values vary depending on the fuzzer used and the metrics selected by the developer of the fuzzer. Table I demonstrates some of the possible metrics used in a fuzzing session, where the "Expected Input" column shows what an application may expect and the "Fuzzed Input" column shows what a fuzzer may generate in its place.

Table I
FUZZING METRIC EXAMPLES

Example	Expected Input	Fuzzed Input
Repeating Delimiters	user@domain.com	user@@domain.com
Using border values	32 bit integer	2^32 + (-1 || +1)
Exceeding expected sizes	Single character	More than 1 or blank input
Incorrect types	Integer	Any non-integer type

Optimising the metrics to use in a fuzzer is important as this can greatly reduce the number of test cases to be executed [1], [5]. Amini and Portnoy emphasises this by demonstrating that if one were to fuzz a single DWORD field in a standard testing environment, it may take approximately 13 years to iterate through all possibilities [6]. To add to this, most protocols consist of more than one field and if we continue to fuzz test a protocol using every single value we can think of, the test becomes uneconomical to run. To streamline the process, we assume that an application is more likely to fail at border cases since, in terms of 32bit integers, an application will likely respond the same to 5367, 5368 and 5369 and issues are more likely to occur when using values around the maximum 32bit, 64bit, 128bit and higher integer ranges. There may be exceptions where an application may specifically looking for one of those values and that value will hence be missed by the fuzzer, but it may not be economical to search for these values depending on the context of the value.

Depending on the quality of the application being tested, fuzzing may cause an application to behave in an unexpected manner and possibly crash it, and it is these crashes that are analysed and manipulated to force the application into an exploitable state. These may manifest as integer and buffer overflow attacks [7]. The goal of fuzz testing is to expose these issues and depending on the motive of the tester, can be beneficial for the application if it is used to expose and fix bugs, or detrimental if the bugs are used to generate exploitable code.

A. Fuzzer Types

Specialised Fuzzers: A specialised fuzzer is one that is purposely built for a specific protocol. There is extensive knowledge on the protocol coded into the application and the fuzzer is aware of specific test-cases that commonly manifest as vulnerabilities in target applications. The advantage of these fuzzers is that they can extensively fuzz an application and focus test cases to a much higher degree [8]. In addition to this, these fuzzers are often deployed with user interfaces that compliment the protocol, such as Infigo FTPStress [8] and Voiper [9].

Fuzzing Frameworks: A fuzzer may not always be available, especially in instances where a user wants to test a proprietary protocol. This is where generic fuzzing frameworks become beneficial. Generic fuzzing frameworks do not target a specific protocol, instead they provide some means of compiling a protocol specification which they then use to fuzz a protocol [10]. This potentially allows for any protocol to be fuzzed, as long as a specification can be built. The benefit of using a fuzzing framework is that it allows a user to build a fuzzer as they need it, and provides a single method of handling fuzz testing sessions across a large scope of protocols [1]. On the other hand, a fuzzing framework relies on the initial specification of the protocol in order to fuzz that protocol correctly, and a mistake in the description could nullify the effects of a fuzzing session [1]. Another disadvantage is the time and complexity of creating the description of a protocol, and even then some frameworks may not have all the necessary requirements and capabilities for modeling every protocol [11]. There are efforts to create fuzzing frameworks that learn protocols autonomously, such as the Evolutionary Fuzzing System [12]. These are however still relatively new and still in development, and thus beyond the scope of this discussion.

Stateless vs Stateful Fuzzers: The ability of the fuzzer to maintain and fuzz a protocol's state can influence the type of bugs found when fuzzing an application. A stateless fuzzer works by manipulating the values in a message and can only fuzz an initial transaction in sequence of transactions. A stateful fuzzer is able to maintain a protocols state and fuzz transactions in any part of a sequence [13]. In [2], the researchers experimented the differences between fuzzing SIP applications using stateless and stateful fuzzers. While several of the target applications were susceptible to stateless fuzzers, some of the applications appeared resilient to fuzzing until a stateful fuzzer was used, where issues that would have otherwise gone unnoticed manifested. This demonstrates the importance of using multiple fuzzing strategies and applications when fuzzing a framework, as one fuzzer may cover test-cases not available in another fuzzer.

B. Testing Strategies

The level of access to the target application also has influence on the strategy of the fuzzing session. These are usually categorised into three types [1]

Whitebox testing: In whitebox testing, the tester has full access to the target application's executable, the source code and has knowledge of the underlying protocol. The benefit of this approach is that a debugger may be attached to the target executable, and the tester is knowledgeable as to what features of the target are more likely to present faults. This can greatly decrease the time it takes to conduct a fuzz test as the generated test cases can be narrowed down. Also, depending on the debugger, extensive reports on crashes may be generated to assist the tester in fixing any issues found. A potential drawback with this approach is that the tester could potentially becomes too focused on testing for a specific flaw and following a specific code path, and could miss less obvious issues in other areas of the application [1]. This scenario is most common in development environments where the source code of an application is readily available.

Greybox testing: Greybox testing is similar to whitebox testing, except the tester has no access to the application's source code. The user has access to the target application's executable and has some knowledge on the workings of the protocol, but any additional information with regards to the applications inner workings is attained through some other means, such as reverse engineering the targets executable

[1]. The benefit of this scenario is that the tester makes no assumptions as to the inner workings of an application and cannot bias the tests toward a specific code path. However the tester is forced to run more tests when compared to whitebox testing as the tests can not be focused. As with the whitebox test, a debugger may be attached to the target application to assist in the fault detection. An example of this scenario is in environments where an end user is evaluating an application, such as an entity evaluating several SIP clients before deciding on one to deploy.

Blackbox testing: In blackbox testing there is no access to an application's source code and access is limited to the application's input and output. There is knowledge on the underlying protocol, but generally this test is conducted over a remote interface, such as when evaluating a remote web server. This forces the tester to use a large range of fuzzing metrics to ensure that the maximum amount of issues are revealed. The benefit of this approach is that the tester has the same view of the target application as the end user, and any issues found are likely to have been experienced by the end user at some point [1]. This method also prevents the testing of any unnecessary execution paths in the application as all the test cases can be considered to be relevant [1]. The issue with this approach is that testing can be much longer than the whitebox and greybox testing approaches, as the test cases developed by the fuzzer are more extensive and can not be focused.

For the purpose of this paper, focus is on the greybox and blackbox approach. This is based on the assumption that the user will always at minimum have some knowledge to the underlying protocol used by an application, and that the user will have some access to an application's executable.

C. Data Generation

Generic fuzzing frameworks can be classified into two main types depending on how they create the malformed queries:

Generation fuzzer: These generate the data used for fuzzing using some sort specification. This specification contains information on the structure of the messages, a specification on the fields within the messages and the sequences of the protocol [14]. The heuristics for generating fuzzed values are determined by the developer of the fuzzer where each variable type will have a specific set of rules, where an integer will be fuzzed differently to a string. Generation fuzzers will usually have some abstract form of representing the data structure of a protocol or format. Figure 1 shows an example of this type of system used by the Sulley fuzzing framework [6]. The benefit of generation fuzzers are that they are able to intelligently determine how to fuzz values, but they are expensive to create as the specification needs to be created [15].

Mutation fuzzer: Mutation fuzzers, instead of learning the structure of a protocol or using a specification, manipulate a valid sample of data on a set of heuristics. These generally fuzz by randomly flipping bits, adding data and manipulating the structure of the sample data [1]. The benefit of this system over that of generation fuzzers is that there is little overhead in setting up the fuzzer, however only trivial cases and protocols are able to be fuzzed since many protocols have checksums to prevent this type of manipulation [1].

```
s_initialize("HTTP VERBS POST")
s_static("POST / HTTP/1.1\r\n")
s_static("Content-Type: ")
s_string("application/x-www-form-urlencoded")
s_static("\r\n")
s_static("Content-Length: ")
s_size("post blob", format="ascii", signed=True,
fuzzable=True)
s_static("\r\n\r\n")
```

Figure 1. Sample of HTTP POST description using Sulley [6]

III. Testing

In this section, the process of performing an actual fuzz test is discussed. Two popular application-layer protocols in use on the Internet were fuzz tested; the File Transfer Protocol (FTP) and the Session Initiation Protocol (SIP). In each instance, several server software implementations were chosen and were then fuzzed using a fuzz testing application built for each individual protocol. The purpose of this analysis is to reinforce the relevance of fuzz testing in software development cycles by testing post-production applications.

For the purpose of this experiment, postmortem debugging is not conducted as the goal is to determine if an application is susceptible to fuzz testing, as apposed to finding specific test-cases that result in an exploitable state. To this end and to test the quality of the fuzzers, only the functionality and completeness of the fuzzers are tested, specifically mentioning the benefits and shortcomings of each fuzzer. This approach also provides an insight into the fuzzing process, and a list of requirements is compiled detailing what components are necessary for a successful fuzzing session.

A. FTP and SIP

The File Transfer Protocol (FTP) is a mature protocol that is simple in structure. It is human readable, consisting of keywords such as STOR (store), RETR (retrieve) and CWD (change working directory), to facilitate the exchange of data over a network [16]. Transactions usually consist of an "Command" and "Variable" pair, such as "USER username" to login a user called "username". According to [16], FTP has four objectives;

- "to promote sharing of files (computer programs and/or data)"
- "to encourage indirect or implicit (via programs) use of remote computers"
- "to shield a user from variations in file storage systems among hosts"
- "to transfer data reliably and efficiently"

To simplify this definition, FTP is used to transfer files across two devices using a single standard.

FTP was chosen as a target protocol due to its simplicity and because of the availability of server and client implementations available on the Internet. In every fuzzing session, the same options were used in both the server applications and in the fuzzing software to ensure consistent results.

SIP is a relatively new protocol when compared to FTP and shares a commonality with FTP where it is human readable using keywords such as INVITE and REGISTER to facilitate communications between applications [16], [17]. SIP acts as a signaling protocol between SIP servers and clients and is usually coupled with other protocols, where it is most commonly used to negotiate streams between these entities. SIP was chosen as the second target as it is a stateful protocol and more complicated that FTP in this regard.

At the time of writing this paper, the vendors of the server applications have not been informed of any issues found as a result of this experimentation. In this respect, the names of the server applications will not be revealed at this time.

B. Fuzzers

To fuzz FTP, Infigo FTPStress [8] and Metasploit [18] was chosen. Infigo FTPStress is a FTP fuzzing application that provides a GUI to simplify the fuzzing process. At the time of testing, it was able to cover 57 FTP commands and allows a user to set default arguments for each of the commands [8]. Metasploit is a penetration testing software suite that consists of a variety of tools, one of which is a simple FTP fuzzing script [18]. Infigo FTPStress was chosen due to its extensive coverage of FTP commands, and Metasploit was selected due to its inclusion in the popular Metasploit suite.

Voiper is a fuzz testing application which uses the Sulley fuzzing framework and focuses specifically on fuzzing the SIP protocol [9]. Voiper uses Sulley to generate test cases and also includes modified modules from Sulley that assist in network and process monitoring, as well as fuzzer automation. Voiper version 0.0.7 was selected as the primary fuzzer for the SIP protocol as it was the most complete SIP fuzzer in terms of scope. It covers the following SIP transactions:

- SIP INVITE
- SIP ACK
- SIP CANCEL
- SIP NOTIFY
- SIP SUBSCRIBE
- SIP REGISTER
- SIP request structure
- SDP over SIP

C. Results

Table II provides a summary of the results found from the fuzzing session against FTP and SIP. FTP showed more favourable results compared to SIP in terms of fuzz testing, but it is assumed that this was because of FTP's simplicity and because two fuzzers were used as apposed to a single fuzzer being used to target the SIP applications. By using multiple fuzzers for FTP, a larger range of test-cases were able to be covered and this possibly lead to the results being biased to FTP.

FTP: Of the 8 FTP server suites tested, only three showed no apparent issues when fuzzed with both Infigo FTPStress and the fuzzing script in Metasploit. All of the remaining five crashed with some sort of non-clean exit code, all with some sort of long string with variations in the delimiters used.

Table II
SUMMARY OF FUZZING SESSIONS ON FTP AND SIP SERVERS

	Servers Tested	**Fuzzers Used**	**Faulted Servers**
FTP	8	2	5
SIP	2	1	0

There was a unique error in one of the server applications where Infigo FTPStress was able to cause the application to access and delete files outside of the allowed location on the hard disk. In three instances, the server applications showed signs of memory leakage as memory usage would deviate from approximately 70mb for normal usage, to 300mb after a fuzz testing session.

Neither Infigo FTPStress nor the Metasploit FTP module provided debugging facilities to automate the fuzzing process. Because of this, whenever a crash occurred the server application had to be manually restarted in order to continue the fuzzing process. This created an overhead as the fuzzing session would stop for a period of time before manual intervention from the tester was required. This overhead extended the time it took for the test to fully be concluded.

SIP: The fuzzing sessions against SIP did not yield any notable results when using Voiper. From the discussion in section II, this may be the result of Voiper not fuzzing the states in a correct manner or Voiper may not have covered enough test-cases to sufficiently test the server applications. Alternatively, the SIP server software that was tested may have been designed in such a way as to be resilient to this type of attack.

The additional Sulley modules built into Voiper provided a high level of automation that allowed a fuzz session to be started and then be left unattended for the during of the test. In addition to this, the process monitor was configured to automatically restart the server application after a certain amount of test cases as to limit the entropy from previous tests.

D. Conclusion

The goal of fuzz testing is to test the robustness of a software implementation and in several instances, the applications showed no signs of crashes or unexpected behaviours. This is not indicative of an application being fully immune to this type of attack since, in the FTP fuzz testing sessions, it was shown that it is possible for an application to have different effects depending on the fuzzer used. This shows that different fuzzers may have different heuristics when generating test cases, and where one fuzzer may miss a crucial test case, another may cover that case. Going by this conclusion, a different SIP fuzzer may have produced different results and potentially have resulted in a SIP server crashing.

Even with FTP being a mature protocol and relatively simple to implement, there are still faulty implementations being developed and deployed on the Internet. As shown here, fuzzing can be used to reveal flaws in applications, even after the development cycle.

IV. Proposed Solution

Using the theory discussed in section II and the analysis in section III, a prototype fuzzing framework has been designed. The goal of this framework is to provide a simplified method for designing protocols and managing generic fuzzing frameworks, such as Peach, Spike and Sulley. In an attempt to minimise errors during the grammar creation process, the framework emphasises on providing a GUI that has some level of affordance, lowering the barrier of entry for creating simple fuzz tests. The framework is based on the Sulley Fuzzing Framework as Sulley's additional functionality beyond that of the actual fuzzing process was found to assist the fuzzing process, including in automating the fuzzing process. Additionally, its open source nature allows for modifications to be done on the underlying framework to better suite the goal of simplifying the fuzzing process. The framework's main goal is to provide a simplified method of fuzzing through an intuitive User Interface using assistive tools and a high level of automation.

A. Framework Structure

The framework is split into three parts; fuzzer manager, protocol design, and an external client. The responsibility of actually fuzzing a protocol and generating test-cases is delegated to a backend framework, such as Peach [19], Spike [20] or Sulley [6]. This has the advantage of relying on the knowledge of researchers that are versed in fuzzing to create a fuzzer, and the responsibility of the maintenance of the fuzzing components is shifted to those researchers. In addition to this, if one framework is unable to sufficiently fuzz a protocol or is missing some feature that is required when fuzzing a specific protocol, the backend may be switched to one that is better suited to that protocol.

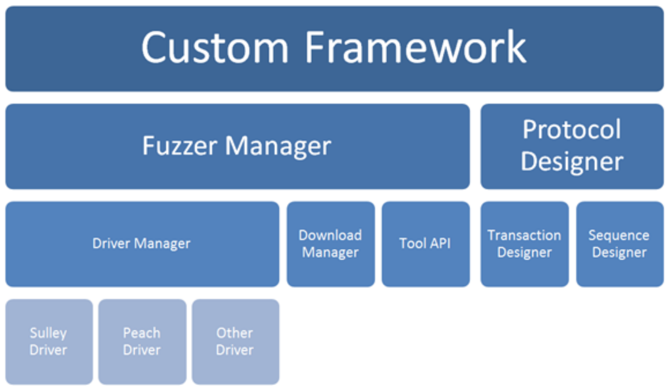

Figure 2. Structure of the Proposed Framework

B. Fuzzer Manager and Client

The fuzzer manager is responsible for selecting the protocol to be fuzzed, controlling the fuzzing session and controlling any additional modules that need to be run, such as the network and process monitor. The fuzzer manager is further divided into three subsections; a driver manager, a tool manager and a download manager.

The conversion from the custom framework's grammar to another format is handled by a driver where each respective framework, such as Sulley, Spike and Peach, has its own driver. The purpose of this driver is to take the grammar generated through the custom framework's user interface and translate it into a format that is readable for a selected framework. The benefit of this is that each driver is loosely coupled with the rest of the framework and a backend framework can be added, modified or removed without affecting the rest of the application.

The tool manager is responsible for controlling any additional features, for simplicity called modules, added to the fuzzer manager. In Figure 3, it is responsible for adding the "Process Monitor", "Network Settings" and any other value adding tabs to the interface. The purpose of the tool manager is to create a single method for controlling any modules added to the framework. It is referred to as the Tool API in Figure 2 since it is designed to behave in a similar fashion to a plugin system, where different modules communicate with the fuzzer manager using a single standard.

The framework has a client application that exists on the target system. The purpose of the client application is to manage the target application and in a similar fashion to Sulley's process monitor, where it controls the target application. It is also used to assist any modules in the fuzzer manager that require some access to the target application.

There are two places that have been identified with the potential to further decrease reliability of the fuzzing process; the protocol design and the translation of the grammar done by the driver. To curb this issue and to further simplify the fuzzing process, a download manager is integrated into the fuzzer manager. The purpose of this component is to promote the sharing of pre-designed protocol designs developed using the framework. This assists in collaboration of protocol design where designers can evaluate and modify other designers work, increasing the quality of the developed fuzzer. The design and usage is based on the Package Manager application featured in the Ubuntu Operating system and works on an HTTP backend. Each protocol design is packaged with the following files:

- a protocol description containing transactions and field formats
- a change log
- a control file, containing all the prerequisites of the fuzzer
- a readme

C. Block-based Protocol Design

The concept of block-based fuzzing, such as in Sulley and Spike, has been taken literally and protocol design is done using block graphics as show in Figure 4. The benefit of this is that it makes the translation from the user interface into a grammar that is recognisable by Sulley and Spike a simple one, where each block in the User Interface represents a line in the Sulley and Spike grammar. The interface has been designed using the QT framework [21] due to its native drag and drop system. The drag and drop functionality allows for a quick method to modifying the protocol design which

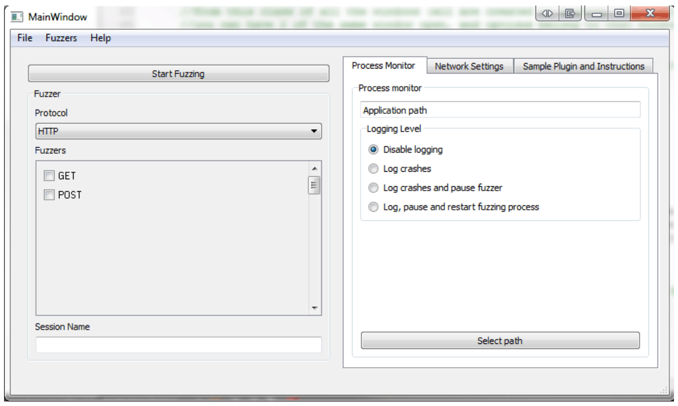

Figure 3. Fuzzer Manager Prototype

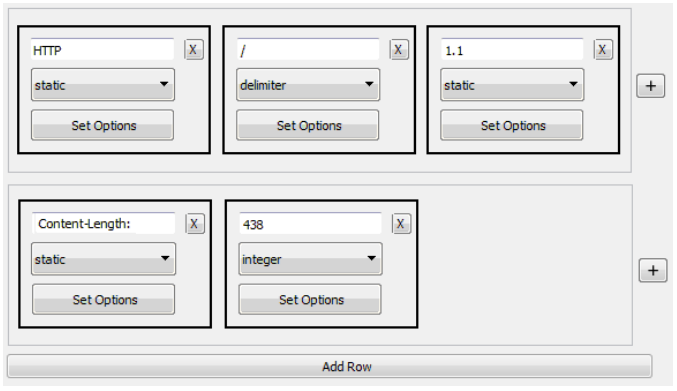

Figure 4. Prototype of the graphical block-based protocol designer

becomes beneficial in environments where a specification of a protocol is under constant revisement.

A packet is described on a field-by-field basis, similar to Sulley and Spike. Every element in a protocol is described in the order that it appears in the protocol, where it is qualified by a series of additional options. In Sulley, as shown in Figure 1 when describing the block “s_size("post blob", format="ascii", signed=True, fuzzable=True)”, the the field s_size is qualified via a set of attributes. In the graphical protocol designer, it is assumed that any field is most commonly featured as a type, represented by a combo box, and a default value, represented in the text edit box. Any other attributes are assumed to be either used for a specific fuzzer or are used to qualify the field. These become difficult to represent as each backend framework may require a different set of options to be specified. Therefore the “Set Options” button is coded to display a table of “Option” and “Value” pairs, where the user can specify what option has what value.

The final stage of protocol design is specifying the sequences of the transactions in a protocol. The graphical framework differs from Sulley in this respect as the convention in Sulley is to store the transactions in the same location as the code that will initiate the fuzzing session, whereas the custom framework stores all aspects of the protocol design in a single file [6]. Figure 5 demonstrates that, from the fuzzers point of view, there are two actors in the fuzzing session; the fuzzer and the target. The user adds a transaction between the targets by pressing the “+” button, where they are then given the option of selecting the direction of the transaction. After this, a combo box, showing all the packets designed in the block-based designer, is displayed and the user selects a packet to transmit. After the transmissions are designed and the protocol design is complete, the user may save the design and the fuzzer creates a text-based representation of the protocol. This text-based representation is the description that will be used by the drivers and converted into a format readable by the backend frameworks.

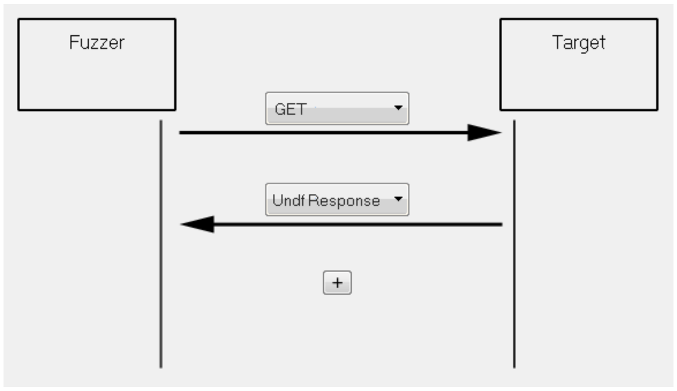

Figure 5. Prototype of the sequence designer

V. Limitations and Future Work

For the initial inception of the framework, the fuzzer driver is compiled with the framework. This makes it difficult to modify the framework when a major change occurs in a backend fuzzing framework, such as a major new release for Sulley. In addition to this the current plugin engine has an API specification, but similar to the driver system, the plugins need to be compiled with the application. A proper plugin system, such as the one employed in the Firefox and Chrome Browsers, will allow 3rd party developers to add additional functionality and fuzzing drivers to the application without the need for the custom fuzzing framework to be recompiled.

Further testing is required in ensuring that the User Interface is sufficient enough for designing any network-based application layer protocol. Both the protocols tested in this paper were simple text-based protocols. There are efforts underway to test a proprietary binary protocol in order to test the effectiveness of the fuzzing framework.

In terms of protocol design, Sulley’s block-based approach is able to design protocols beyond that of application layer network protocols. The framework could potentially be modified to allow for different mediums to be designed and fuzzed.

VI. Conclusion

Work done by [2] and [4], and the analysis in section III shows that fuzzing can be an effective method for finding bugs in applications. These bugs may manifest as exploitable code, therefore it is in the best interest of the developers to fuzz test their applications before deployment. Despite this, fuzz testing is not very prevalent in development environments and the assumption is made that the difficulty in conducting fuzz tests deters developers from conducting these tests.

A fuzzing framework featuring a User Interface was designed. The was to provide a single means of designing a large variety of protocols, hence enforcing the philosophy of learn once, use multiple times. It is believed that the framework will allow for simple application-layer protocols to be designed and tested in a simpler method when compared to traditional fuzzing frameworks, although more complex protocols may not be able to be sufficiently covered by the framework without further testing and modifications.

REFERENCES

[1] P. Amini, A. Greene, and M. Sutton, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[2] H. J. Abdelnur, R. State, and O. Festor, "Kif: A stateful sip fuzzer," in *IPTCOMM, New York USA*, 2007.

[3] C. Mulliner and C. Miller, "Injecting sms messages into smart phones for security analysis," 2009.

[4] (2011, May) The trustworthy computing security development lifecycle. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms995349.aspx

[5] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing." Network and Distributed System Security Symposium, 2008.

[6] P. Amini and A. Portnoy, *Sulley: Fuzzing Framework*, 2008.

[7] J. Seitz, *Gray Hat Python*. William Pollock, 2009.

[8] L. Juranic, "Using fuzzing to detect security vulnerabilities," 2006.

[9] (2012, January) Voiper. [Online]. Available: http://voiper.sourceforge.net/

[10] P. Amini and A. Portnoy, "Fuzzing frameworks," in *Blackhat*, 2007. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-07/Amini_and_Portnoy/Whitepaper/bh-usa-07-amini_and_portnoy-WP.pdf

[11] A. Takanen, J. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., 2008.

[12] J. Godinez and R. Mortam, "Evolutionary fuzzing system," 2010.

[13] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: toward a stateful network protocol fuzzer," 2007.

[14] C. Miller, "How smart is intelligent fuzzing -or- how stupid is dumb fuzzing?" 2005.

[15] C. Miller and Z. N. J. Peterson, "Analysis of mutation and generation-based fuzzing," March 2007.

[16] J. Postel and J. Reynolds. (2012, April) File transfer protocol rfc. Network Working Group. [Online]. Available: http://www.ietf.org/rfc/rfc959.txt

[17] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. (2012, April) Session initiation protocol rfc. Network Working Group. [Online]. Available: http://tools.ietf.org/rfc/rfc3665.txt

[18] (2011, June) Metasploit framework. [Online]. Available: http://www.metasploit.com/

[19] (2012, April) Peach fuzzing platform. [Online]. Available: http://peachfuzzer.com/

[20] (2012, May) Spike fuzzer. [Online]. Available: http://immunityinc.com/resources-freesoftware.shtml

[21] (2012, April) Qt cross-platform application and ui framework. Nokia. [Online]. Available: http://qt.nokia.com/