Fuzzing Frameworks

Pedram Amini <pamini@fuzzing.org>

Introduction

There are a number of available specialized fuzzing utilities which target many common and documented network protocols and file formats. These fuzzers exhaustively iterate through a designated protocol and can be used across the board to stress test a variety of applications that support that protocol. For instance, the same specialized SMTP fuzzer could be used against a variety of e-mail transfer programs such as Microsoft Exchange, Sendmail, qmail, etc. Other "dumb" fuzzers take a more generic approach to allow for fuzzing of arbitrary protocols and file formats and perform simple, non-protocol-aware mutations such as bit flipping and byte transposing. Although these fuzzers are effective against a wide range of common applications, we often have a need for more customization and thorough fuzzing for proprietary and previously untested protocols. This is where fuzzing frameworks become extremely useful.

In this paper, we explore a number of open source fuzzing frameworks available today, including SPIKE, the ever popular framework which has become a household name (depending on how geeky your household is). We also look at some exciting newcomers in the field such as Autodafé and GPF. Following the dissection of the existing technologies we then see how, despite the power supplied by many general-purpose fuzzing frameworks, we will still need to create a fuzzer from scratch once in a while. We'll illustrate this point later with a real-world example fuzzing problem and the development of a solution. Finally, we introduce a new framework released at BlackHat US 2007 and explore the advances made by the effort.

This paper is an excerpt from "Fuzzing: Brute Force Vulnerability Discovery"[1] published by Addison-Wesley Professional.

What Is a Fuzzing Framework?

Some of the fuzzing frameworks available today are developed in C, while others in Python or Ruby. Some offer functionality in their native language, whereas others leverage a custom language. For instance, the Peach fuzzing framework exposes constructs in Python, while dfuz implements its own set of fuzzing objects (both of these frameworks are discussed in more detail later). Some abstract data generation and others don't. Some are object oriented and well documented; others are usable for the most part only by the creator. However, the common goal of all fuzzing frameworks is the same; to provide a quick, flexible, reusable, and homogenous development environment to fuzzer developers.

A good fuzzing framework should abstract and minimize a number of tedious tasks. To assist with the first stages of protocol modeling, some frameworks include utilities for converting captured network traffic into a format understandable by the framework. Doing so allows a researcher to import large chunks of empirical data and focus his efforts on a more human-suitable task such as determining protocol field boundaries.

Automatic length calculation is an absolute necessity for a well-rounded framework. Many protocols are implemented using a TLV (Type, Length, Value) style syntax, similar to the ASN.1[2]

[1] http://tinyurl.com/3akcqb (ISBN: 0321446119)

[2] http://en.wikipedia.org/wiki/Asn.1

standard. Consider an example where the first byte of data communication defines the *type* of data to follow: 0x01 for plain text and 0x02 for raw binary. The next two bytes define the *length* of the data to follow. Finally, the remaining bytes define the *value*, or the data, specific to the communication as shown here:

01	00 07	F U Z Z I N G
Type	Length	Value

When fuzzing the Value field of this protocol, we must calculate and update the two-byte length field in every test case. Otherwise, we risk our test cases getting immediately dropped if the communication is detected as breaching protocol specifications. Calculating Cyclic Redundancy Check (CRC)[3] calculations and other checksum algorithms are other tasks a useful framework should include. CRC values are commonly found embedded in both file and protocol specifications to identify potentially corrupted data. PNG image files, for example, employ CRC values that allow programs to avoid processing an image if the received CRC does not match the calculated value. Although this is an important feature for security and functionality, it will prohibit fuzzing efforts if the CRC is not correctly updated as a protocol is mutated. As a more extreme example, consider the Distributed Network Protocol (DNP3)[4] specification, which is utilized in Supervisory Control and Data Acquisition (SCADA) communications. Data streams are individually sliced into 250-byte chunks and each chunk is prefixed with a CRC-16 checksum! Finally, consider that the IP addresses of the client, server, or both are frequently seen within transmitted data and that both addresses might change frequently during the course of a fuzz test. It would be convenient for a framework to offer a method of automatically determining and including these values in your generated fuzz test cases.

Most, if not all, frameworks provide methods for generating pseudo-random data. A good framework will take a further step by including a strong list of attack heuristics. An attack heuristic is nothing more than a stored sequence of data that has previously been known to cause a software fault. Format string (%n%n%n%n) and directory traversal (../../../) sequences are common examples of simple attack heuristics. Cycling through a finite list of such test cases prior to falling back on random data generation will save time in many scenarios and is a worthwhile investment.

Fault detection plays an important role in fuzzing. A fuzzer can detect that its target might have failed at the most simple level if the target is unable to accept a new connection. More advanced fault detection is generally implemented with the assistance of a debugger. An advanced fuzzing framework should allow the fuzzer to directly communicate with a debugger attached to the target application or even bundle a custom debugger technology altogether.

Depending on your personal preference, there may be a long laundry list of minor features that can significantly improve your fuzzer development experience. For example, some frameworks include support for parsing a wide range of formatted data. When copying and pasting raw bytes into a fuzzing script, for example, it would be convenient to be able to paste hex bytes in any of the following formats: 0x41 0x42, \x41 \x42, 4142, and so on.

Fuzzing metrics have also received little attention to date. An advanced fuzzing framework may include an interface for communicating with a metric gathering tool, such as a code coverage monitor.

Finally, the ideal fuzzing framework will provide facilities that maximize code reuse by making developed components readily available for future projects. If implemented correctly, this concept allows a fuzzer to evolve and get “smarter” the more it is used. Keep these concepts in mind as we explore a number of fuzzing frameworks prior to delving into the design and creation of both a task-specific and general-purpose custom framework.

[3] http://en.wikipedia.org/wiki/Cyclic_redundancy_check
[4] http://www.dnp.org/

Existing Frameworks

In this section, we dissect a number of fuzzing frameworks to gain an understanding of what already exists. We do not cover every available fuzzing framework, but instead, we examine a sampling of frameworks that represent a range of different methodologies. The following frameworks are listed in a general order of maturity and feature richness, starting with the most primitive.

antiparser[5]

Written in Python, antiparser is an API designed to assist in the creation of random data specifically for the construction of fuzzers. This framework can be used to develop fuzzers that will run across multiple platforms as the framework depends solely on the availability of a Python interpreter. Use of the framework is pretty straightforward. An instance of the antiparser class must first be created; this class serves as a container. Next, antiparser exposes a number of fuzz types that can be instantiated and appended to the container. Available fuzz types include the following:

- apChar(): An eight-bit C character.
- apCString(): A C-style string; that is, an array of characters terminated by a null byte.
- apKeywords(): A list of values each coupled with a separator, data block, and terminator.
- apLong(): A 32-bit C integer.
- apShort(): A 16-bit C integer.
- apString(): A free form string.

Of the available data types, apKeywords() is the most interesting. With this class, you define a list of keywords, a block of data, a separator between keywords and the data, and finally an optional data block terminator. The class will generate data in the format [keyword] [separator] [data block] [terminator].

antiparser distributes an example script, evilftpclient.py, that leverages the apKeywords() data type. Let's examine portions of the script to gain a better understanding of what development on this framework looks like. The following Python code excerpt shows the relevant portions of evilftpclient.py responsible for testing an FTP daemon for format string vulnerabilities in the parsing of FTP verb arguments. This excerpt does not display the functionality for authenticating with the target FTP daemon, for example. For a complete listing, refer to the source.

```
from antiparser import *

CMDLIST = ['ABOR', 'ALLO', 'APPE', 'CDUP', 'XCUP', 'CWD',
           'XCWD', 'DELE', 'HELP', 'LIST', 'MKD',  'XMKD',
           'MACB', 'MODE', 'MTMD', 'NLST', 'NOOP', 'PASS',
           'PASV', 'PORT', 'PWD',  'XPWD', 'QUIT', 'REIN',
           'RETR', 'RMD',  'XRMD', 'REST', 'RNFR', 'RNTO',
           'SITE', 'SIZE', 'STAT', 'STOR', 'STRU', 'STOU',
           'SYST', 'TYPE', 'USER']

SEPARATOR  = " "
TERMINATOR = "\r\n"
```

[5] http://antiparser.sourceforge.net/

```
for cmd in CMDLIST:
    ap = antiparser()
    cmdkw = apKeywords()
    cmdkw.setKeywords([cmd])
    cmdkw.setSeparator(SEPARATOR)
    cmdkw.setTerminator(TERMINATOR)

    cmdkw.setContent(r"%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n")
    cmdkw.setMode('incremental')
    cmdkw.setMaxSize(65536)
    ap.append(cmdkw)

    sock = apSocket()
    sock.connect(HOST, PORT)

    # print FTP daemon banner
    print sock.recv(1024)

    # send fuzz test
    sock.sendTCP(ap.getPayload())

    # print FTP daemon response
    print sock.recv(1024)
    sock.close()
```

The code begins by importing all the available data types and classes from the antiparser framework, defining a list of FTP verbs, the verb argument separator character (space), and the command termination sequence (carriage return followed by newline). Each listed FTP verb is then iterated and tested separately, beginning with the instantiation of a new antiparser container class. Next, the apKeywords() data type comes into play. A list defining a single item, the current verb being tested, is specified as the keywords (keyword in this case). The appropriate verb[nd]argument separation and command termination characters are then defined. The data content for the created apKeyword() object is set to a sequence of format string tokens. If the target FTP server exposes a format string vulnerability in the parsing of verb arguments, this data content will surely trigger it.

The next two calls, setMode('incremental') and setMaxSize(65536), specify that on permutation the data block should grow sequentially to the maximum value of 65,336. However, in this specific case, those two calls are irrelevant, as the fuzzer does not loop through a number of test cases or permutations by calling ap.permute(). Instead, each verb is tested with a single data block.

The remaining lines of code are mostly self-explanatory. The single data type, apKeywords(), is appended to the antiparser container and a socket is created. Once a connection has been established, the test case is generated in the call to ap.getPayload() and transmitted via sock.sendTCP().

It is obvious that antiparser has a number of limitations. The example FTP fuzzer can be easily reproduced in raw Python without assistance from the framework. The ratio of framework-specific code versus generic code when developing fuzzers on antiparser is significantly low in comparison to other frameworks. The framework also lacks many of the desired automations listed in the previous section, such as the ability to automatically calculate and represent the common TLV protocol format. On a final note, the documentation for this framework is slim and, unfortunately, only a single example is available as of version 2.0, which was released in August 2005. Although this framework is simple and provides some benefits for generating simple fuzzers, it is inadequate for handling complex tasks.

Dfuz[6]

Written in C by Diego Bauche, Dfuz is an actively maintained and frequently updated fuzzing framework. This framework has been used to uncover a variety of vulnerabilities affecting vendors such as Microsoft, Ipswitch, and RealNetworks. The source code is open and available for download but a restrictive license prevents any duplication or modification without express permission of the author. Depending on your needs, this limited license may dissuade you from utilizing the framework. The motivating factor behind this rigid licensing appears to be that the author is unhappy with the quality of his own code (according to the README file). It might be worth a shot to contact him directly if you wish to reuse portions of his code. Dfuz was designed to run on UNIX/Linux and exposes a custom language for developing new fuzzers. This framework is not the most advanced fuzzing framework dissected in this paper. However, its simple and intuitive design makes it a good framework design case study, so let's examine it in detail.

The basic components that comprise Dfuz include data, functions, lists, options, protocols, and variables. These various components are used to define a set of rules that the fuzzing engine can later parse to generate and transmit data. The following familiar and simple syntax is used to define variables within rule files:

```
var my_variable = my_data
var ref_other   = "1234",$my_variable,0x00
```

Variables are defined with the simple prefix *var* and can be referenced from other locations by prefixing the variable name with a dollar sign ($, like in Perl or PHP). Fuzzer creation is entirely self-contained. This means that unlike antiparser, for example, building a fuzzer on top of this framework is done entirely in its custom scripting language.

Dfuz defines various functions to accomplish frequently needed tasks. Functions are easily recognizable as their names are prefixed with the percent character (%). Defined functions include the following:

- %attach(): Wait until the Enter or Return key is pressed. This is useful for pausing the fuzzer to accomplish another task. For example, if your fuzzing target spawns a new thread to handle incoming connections and you wish to attach a debugger to this new thread, insert a call to %attach() after the initial connection is established, then locate and attach to the target thread.
- %length() or %calc(): Calculate and insert the size of the supplied argument in binary format. For example, %length("AAAA") will insert the binary value 0x04 into the binary stream. The default output size for these functions is 32 bits, but this can be modified to 8 bits by calling %length:uint8() or to 16 bits by calling %length:uint16().
- %put:<size>(number) : Insert the specified number into the binary stream. The size can be specified as one of byte, word, or dword, which are aliases of uint8, uint16, and uint32, respectively.
- %random:<size>(): Will generate and insert a random binary value of the specified size. Similar to %put(), size can be specified as one of byte, word, or dword, which are aliases of uint8, uint16, and uint32, respectively.
- %random:data(<length>,<type>): Generate and insert random data. Length specifies the number of bytes to generate. Type specifies the kind of random data to generate and can be specified as one of ascii, alphanum, plain, or graph.
- %dec2str(num) : Convert a decimal number to a string and insert it into the binary stream. For example, %dec2str(123) generates 123.
- %fry(): Randomly modify the previously defined data. The rule "AAAA",%fry(), for

[6] http://www.genexx.org/dfuz/

example, will cause a random number of the characters in the string "AAAA" to be replaced with a random byte value.

- %str2bin(): Parses a variety of hexadecimal string representations into their raw value. For example: 4141, 41 41, and 41-41 would all translate to AA.

Data can be represented in a number of ways. The custom scripting language supports syntax for specifying strings, raw bytes, addresses, data repetitions, and basic data looping. Multiple data definitions can be strung together to form a simple list by using the comma separator. The following examples demonstrate a variety of ways that data can be defined (refer to the documentation for further details):

```
var my_variable1 = "a string"
var my_variable2 = 0x41,|0xdeadbeef|,[Px50],[\x41*200],100
```

Lists are declared with the keyword *list* followed by the list name, the keyword *begin*, a list of data values separated by newlines, and are finally terminated with the keyword *end*. A list can be used to define and index a sequence of data. For example:

```
list my_list:
begin
    some_data
    more_data
    even_more_data
end
```

Like variables, lists can be referenced from other locations by prefixing the list name with a dollar sign ($). With similar syntax to other scripting languages, such as Perl and PHP, a list member can be indexed through square brackets: $my_list[1]. Random indexes within a list are also supported through the *rand* keyword: $my_list[rand].

A number of options for controlling the behavior of the overall engine exist, including the following:

- *keep_connecting*. Continue fuzzing the target even if a connection cannot be established.
- *big_endian*. Changes the byte order of data generation to big endian (little endian is the default).
- *little_endian*. Changes the byte order of data generation to little endian (the default).
- *tcp*. Specifies that socket connections should be established over TCP.
- *udp*. Specifies that socket connections should be established over UDP.
- *client_side*. Specifies that the engine will be fuzzing a server and thereby acting as a client.
- *server_side*. Specifies that the engine will be fuzzing a client and thereby acting as a server waiting for a connection.
- *use_stdout*. Generate data to standard output (console) as opposed to a socket-connected peer. This option must be coupled with a host value of “stdout.”

To ease the burden of reproducing frequently fuzzed protocols, Dfuz can emulate the FTP, POP3, Telnet, and Server Message Block (SMB) protocols. This functionality is exposed through functions such as ftp:user(), ftp:pass(), ftp:mkd(), pop3:user(), pop3:pass(), pop3:dele(), telnet:user(), telnet:pass(), smb:setup(), and so on. Refer to the Dfuz documentation for a complete listing.

These basic components must be combined with some additional directives to create rule files. As a simple, yet complete example, consider the following rule file (bundled with the framework) for fuzzing an FTP server:

```
port=21/tcp

peer write: @ftp:user("user")
peer read
peer write: @ftp:pass("pass")
peer read
peer write: "CWD /", %random:data(1024,alphanum), 0x0a
peer read
peer write: @ftp:quit()
peer read

repeat=1024
wait=1
# No Options
```

The first directive specifies that the engine must connect over TCP port 21. As no options are specified, it defaults to behaving as a client. The peer read and peer write directives indicate to the engine when data should be read from and written to the fuzz target, respectively. In this specific rule file, the FTP protocol functionality is used to authenticate with the target FTP server. Next, the Change Working Directory (CWD) command is manually constructed and transmitted to the server. The CWD command is fed 1,024 random bytes of alphanumeric data followed by a terminating newline (0x0a). Finally the connection is closed. The final repeat directive specifies that the peer read and write block should be executed 1,024 times. With every test case, Dfuz will establish an authenticated connection with the FTP server, issue a CWD command with a random 1,024-byte alphanumeric string as the argument, and tear down the connection.

Dfuz is a simple and powerful fuzzing framework that can be used to replicate and fuzz many protocols and file formats. The combination of stdout (standard output) support with some basic command-line scripting can transform this framework into a file format, environment variable, or command-line argument fuzzer as well. Dfuz has a relatively quick learning curve and fast development time. The fact that fuzzer development is accomplished entirely in its own scripting language is a double-edged sword. It is a positive in that nonprogrammers can describe and fuzz protocols on this framework and it is a negative in that experienced programmers cannot leverage the innate powers and features exposed by a mature programming language. Dfuz promotes some code reuse, but not nearly as much as other frameworks, such as Peach. A key feature currently lacking is the availability of an intelligent set of attack heuristics. Overall, Dfuz is an interesting case study for a well-designed fuzzing framework and a good tool to keep in the back pocket.

SPIKE[7]

SPIKE, written by Dave Aitel, is probably the most widely used and recognized fuzzing framework. SPIKE is written in C and exposes an API for quickly and efficiently developing network protocol fuzzers. SPIKE is open source and released under the flexible GNU General Public License (GPL).[8] This favorable licensing has allowed for the creation of SPIKEfile[9], a repurposed version of the framework designed specifically for file format. SPIKE utilizes a novel technique for representing and thereafter fuzzing network protocols. Protocol data structures are broken down and represented as blocks, also referred to as a SPIKE, which contains both binary

[7] http://www.immunitysec.com/resources-freesoftware.shtml
[8] http://www.gnu.org/copyleft/gpl.html
[9] http://www.fuzzing.org

data and the block size. Block-based protocol representation allows for abstracted construction of various protocol layers with automatic size calculations. To better understand the block-based concept, consider the following simple example from the whitepaper “The Advantages of Block-Based Protocol Analysis for Security Testing”:[10]

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_block_end("somepacketdata");
```

This basic SPIKE script (SPIKE scripts are written in C) defines a block named somepacketdata, pushes the four bytes 0x01020304 into the block and prefixes the block with the block length. In this case the block length would be calculated as 4 and stored as a big endian word. Note that most of the SPIKE API is prefixed with either s_ or spike_. The s_binary() API is used to add binary data to a block and is quite liberal with its argument format, allowing it to handle a wide variety of copied and pasted inputs such as the string 4141 \x41 0x41 41 00 41 00. Although simple, this example demonstrates the basics and overall approach of constructing a SPIKE. As SPIKE allows blocks to be embedded within other blocks, arbitrarily complex protocols can be easily broken down into their smallest atoms. Expanding on the previous example:

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_blocksize_halfword_bigendian("innerdata");
s_block_start("innerdata");
s_binary("00 01");
s_binary_bigendian_word_variable(0x02);
s_string_variable("SELECT");
s_block_end("innerdata");
s_block_end("somepacketdata");
```

In this example, two blocks are defined, somepacketdata and innerdata. The latter block is contained within the former block and each individual block is prefixed with a size value. The newly defined innerdata block begins with a static two-byte value (0x0001), followed by a four-byte variable integer with a default value of 0x02, and finally a string variable with a default value of SELECT. The s_binary_bigendian_word_variable() and s_string_variable() APIs will loop through a predefined set of integer and string variables (attack heuristics), respectively, that have been known in the past to uncover security vulnerabilities. SPIKE will begin by looping through the possible word variable mutations and then move on to mutating the string variable. The true power of this framework is that SPIKE will automatically update the values for each of the size fields as the various mutations are made. To examine or expand the current list of fuzz variables, look at SPIKE/src/spike.c. Version 2.9 of the framework contains a list of almost 700 error-inducing heuristics.

Using the basic concepts demonstrated in the previous example, you can begin to see how arbitrarily complex protocols can be modeled in this framework. A number of additional APIs and examples exist. Refer to the SPIKE documentation for further information. Sticking to the running example, the following code excerpt is from an FTP fuzzer distributed with SPIKE. This is not the best showcase of SPIKE’s capabilities, as no blocks are actually defined, but it helps to compare apples with apples.

```
s_string("HOST ");
s_string_variable("10.20.30.40");
s_string("\r\n");
```

[10] http://www.immunitysec.com/downloads/advantages_of_block_based_analysis.pdf

```
s_string_variable("USER");
s_string(" ");
s_string_variable("bob");
s_string("\r\n");
s_string("PASS ");
s_string_variable("bob");
s_string("\r\n");

s_string("SITE ");
s_string_variable("SEDV");
s_string("\r\n");

s_string("ACCT ");
s_string_variable("bob");
s_string("\r\n");

s_string("CWD ");
s_string_variable(".");
s_string("\r\n");

s_string("SMNT ");
s_string_variable(".");
s_string("\r\n");

s_string("PORT ");
s_string_variable("1");
s_string(",");
s_string_variable("2");
s_string(",");
s_string_variable("3");
s_string(",");
s_string_variable("4");
s_string(",");
s_string_variable("5");
s_string(",");
s_string_variable("6");
s_string("\r\n");
```

SPIKE is sporadically documented and the distributed package contains many deprecated components that can lead to confusion. However, a number of working examples are available and serve as excellent references for familiarizing with this powerful fuzzing framework. The lack of complete documentation and disorganization of the distribution package has led some researchers to speculate that SPIKE is purposefully broken in a number of areas to prevent others from uncovering vulnerabilities privately discovered by the author. The veracity of this claim remains unverified.

Depending on your individual needs, one major pitfall of the SPIKE framework is the lack of support for Microsoft Windows, as SPIKE was designed to run in a UNIX environment, although there are mixed reports of getting SPIKE to function on the Windows platform through Cygwin.[11] Another factor to consider is that even minor changes to the framework, such as the addition of new fuzz strings, require a recompilation. On a final negative note, code reuse between developed fuzzers is a manual copy-and-paste effort. New elements such as a fuzzer for e-mail addresses cannot simply be defined and later referenced globally across the framework.

[11] http://www.cygwin.com/

Overall, SPIKE has proven to be effective and has been used by both its author and others to uncover a variety of high-profile vulnerabilities. SPIKE also includes utilities such as a proxy, allowing a researcher to monitor and fuzz communications between a browser and a Web application. SPIKE's fault-inducing capabilities have gone a long way in establishing the value of fuzzing on a whole. The block-based approach to fuzzing has gained popularity evident in that since the initial public release of SPIKE, a number of fuzzing frameworks have adopted the technique.

Peach[12]

Peach, released by IOACTIVE, is a cross-platform fuzzing framework written in Python and originally released in 2004. Peach is open source and openly licensed. Compared with the other available fuzzing frameworks, Peach's architecture is arguably the most flexible and promotes the most code reuse. Furthermore, it arguably has the most interesting name (peach "fuzz" get it?). The framework exposes a number of basic components for constructing new fuzzers, including generators, transformers, protocols, publishers, and groups.

Generators are responsible for generating data ranging from simple strings to complex layered binary messages. Generators can be chained together to simplify the generation of complex data types. Abstraction of data generation into its own object allows for easy code reuse across implemented fuzzers. Consider, for example, that an e-mail address generator was developed during an SMTP server audit. That generator can be transparently reused in another fuzzer that requires generation of e-mail addresses.

Transformers change data in a specific way. Example transformers might include a base64 encoder, gzip, and HTML encoding. Transformers can also be chained together and can be bound to a generator. For example, a generated e-mail address can be passed through a URL-encoding transformer and then again through a gzip transformer. Abstraction of data transformation into its own object allows for easy code reuse across implemented fuzzers. Once a given transformation is implemented, it can be transparently reused by all future developed fuzzers.

Publishers implement a form of transport for generated data through a protocol. Example publishers include file publishers and TCP publishers. Again, the abstraction of this concept into its own object promotes code reuse. Although not possible in the current version of Peach, the eventual goal for publishers is to allow transparent interfacing with any publisher. Consider, for example, that you create a GIF image generator. That generator should be able to publish to a file or post to a Web form by simply swapping the specified publisher.

Groups contain one or more generators and are the mechanism for stepping through the values that a generator is capable of producing. Several stock group implementations are included with Peach. An additional component, the Script object, is a simple abstraction for reducing the amount of redundant code required for implementing looping through data through calls to group.next() and protocol.step().

As a complete, but simple example, consider the following Peach fuzzer designed to brute force the password of an FTP user from a dictionary file:

```
from Peach              import *
from Peach.Transformers import *
from Peach.Generators   import *
from Peach.Protocols    import *
from Peach.Publishers   import *

loginGroup = group.Group()
loginBlock = block.Block()
loginBlock.setGenerators((
```

[12] http://www.ioactive.com/tools.html

```
    static.Static("USER username\r\nPASS "),
    dictionary.Dictionary(loginGroup, "dict.txt"),
    static.Static("\r\nQUIT\r\n")
    ))

loginProt = null.NullStdout(ftp.BasicFtp('127.0.0.1', 21), loginBlock)


script.Script(loginProt, loginGroup, 0.25).go()
```

The fuzzer begins by importing the various components of the Peach framework. Next, a new block and group is instantiated. The block is defined to pass a username and then the verb of the password command. The next element of the block imports a dictionary of potential passwords. This is the block element that will be iterated during fuzzing. The final element of the block terminates the password command and issues the FTP quit command to disconnect from the server. A new protocol is defined, extending from the already available FTP protocol. Finally, a script object is created to orchestrate the looping of connections and iterations through the dictionary. The first thought that might come to mind after looking over this script is that interfacing with the framework is not very intuitive. This is a valid criticism and probably the biggest pitfall of the Peach framework. Developing your first fuzzer in Peach will definitely take you longer than developing your first fuzzer in Autodafé or Dfuz.

The Peach architecture allows a researcher to focus on the individual subcomponents of a given protocol, later tying them together to create a complete fuzzer. This approach to fuzzer development, although arguably not as fast as the block-based approach, certainly promotes the most code reuse of any other fuzzing framework. For example, if a gzip transformer must be developed to test an antivirus solution, then it becomes available in the library for transparent use later on to test an HTTP server's capability of handling compressed data. This is a beautiful facet of Peach. The more you use it, the smarter it gets. Thanks to its pure Python implementation, a Peach fuzzer can be run from any environment with a suitable Python installation. Furthermore, by leveraging existing interfaces such as Python's, Microsoft's COM[13], or Microsoft .NET packages, Peach allows for direct fuzzing of ActiveX controls and managed code. Examples are also provided for directly fuzzing Microsoft Windows DLLs as well as embedding Peach into C/C++ code for creating instrumented clients and servers.

Peach is under active development and as of the time of publication the latest available version is 0.5 (released in April 2006). Although Peach is highly advanced in theory, it is unfortunately not thoroughly documented, nor is it widely used. This lack of reference material results in a substantial learning curve that might dissuade you from considering this framework. The author of Peach has introduced some novel ideas and created a strong foundation for expansion. On a final note, a Ruby port of the Peach framework has been announced, although no further details were available at of the time of writing.

General Purpose Fuzzer[14]

General Purpose Fuzzer (GPF), released by Jared DeMott of Applied Security, is named as a play on words on the commonly recognized term general protection fault. GPF is actively maintained, available as open source under the GPL license, and is developed to run on a UNIX platform. As the name implies, GPF is designed as a generic fuzzer; unlike SPIKE, it can generate an infinite number of mutations. This is not to say that generation fuzzers are superior to heuristic fuzzers, as both methodologies have pros and cons. The major advantage of GPF over the other frameworks listed in this section is the low cost of entry in getting a fuzzer up and running. GPF exposes functionality through a number of modes, including PureFuzz, Convert, GPF (main mode), Pattern Fuzz, and SuperGPF.

[13] http://en.wikipedia.org/wiki/Component_Object_Model
[14] http://www.appliedsec.com/developers.html

PureFuzz is an easy-to-use purely random fuzzer, similar to attaching /dev/urandom to a socket. Although the generated input space is unintelligent and infinite, this technique has discovered vulnerabilities in the past, even in common enterprise software. The main advantage of PureFuzz over the netcat and /dev/urandom combination is that PureFuzz exposes a seed option allowing for pseudo-random stream replay. Furthermore, in the event that PureFuzz is successful, the specific packets responsible for causing an exception can be pinpointed through the use of a range option.

Convert is a GPF utility that can translate libpcap files, such as those generated by Ethereal[15] and Wireshark,[16] into a GPF file. This utility alleviates some tedium from the initial stages of protocol modeling by converting the binary pcap (packet capture) format into a human-readable and ready-to-modify text-based format.

In GPF's main mode, a GPF file and various command-line options are supplied to control a variety of basic protocol attacks. Captured traffic is replayed, portions of which are mutated in various ways. Mutations include insertion of progressively larger string sequences and format string tokens, byte cycling, and random mutations, among others. This fuzzing mode is manually intensive as the bulk of its logic is built on the analyst's instinct.

Pattern Fuzz (PF) is the most notable GPF mode due to its ability to automatically tokenize and fuzz detected plain text portions of protocols. PF examines target protocols for common ASCII boundaries and field terminators and automatically fuzzes them according to set of internal rules. The internal rules are defined in C code named tokAids. The ASCII mutation engine is defined as a tokAid (normal_ascii) and there are a few others (e.g., DNS). To accurately and intelligently model and fuzz a custom binary protocol, a tokAid must be written and compiled.

SuperGPF is a Perl script GPF wrapper written to address situations where a specific socket endpoint has been targeted for fuzzing but the researcher has no idea where to begin. SuperGPF combines a GPF capture with a text file containing valid protocol commands and generates thousands of new capture files. The script then starts multiple instances of GPF in varying fuzz modes to bombard the target with a wide variety of generated data. SuperGPF is limited to the fuzzing of ASCII protocols only.

Again, we provide an example GPF script to compare and contrast with previously shown FTP fuzzers:

```
Source:S Size:20 Data:220 (vsFTPd 1.1.3)
Source:C Size:12 Data:USER jared
Source:S Size:34 Data:331 Please specify the password.
Source:C Size:12 Data:PASS jared
Source:S Size:33 Data:230 Login successful. Have fun.
Source:C Size:6 Data:QUIT
Source:S Size:14 Data:221 Goodbye.
```

The biggest setback to working with GPF is its complexity. The learning curve required to get started with GPF is significant. Modeling proprietary binary protocols with tokAids is not as simple as other alternatives, such as SPIKE's block-based approach, and furthermore requires compilation to use. Finally, a heavy dependence on command-line options results in unwieldy commands such as the following:

```
GPF ftp.gpf client localhost 21 ? TCP 8973987234 100000 0 + 6 6 100 100
5000 43 finsih 0 3 auto none -G b
```

Overall, GPF is a valuable tool due to both its flexibility and extensibility. The varying modes allow a researcher to start fuzzing quickly while working on setting up a second and more intelligent wave of attacks. The capability to automatically process and fuzz ASCII protocols is

[15] http://www.ethereal.com
[16] http://www.wireshark.org

powerful and stands out among other frameworks. As of the time of writing, the author of GPF is currently developing a new fuzzing mode that leverages the fundamentals of evolutionary computing to automatically dissect and intelligently fuzz unknown protocols.

Autodafé[17]

Quite simply, Autodafé can be described as the next generation of SPIKE and can be used to fuzz both network protocols and file formats. Autodafé is released under the flexible GNU GPL by Martin Vuagnoux and is designed to run on UNIX platforms. Similar to SPIKE, Autodafé's core fuzzing engine takes a block-based approach for protocol modeling. The following excerpt from the whitepaper "Autodafé, an Act of Software Torture"[18] demonstrates the block-based language used by Autodafé and will look familiar to SPIKE users:

```
string("dummy");             /* define a constant string */
string_uni("dummy");         /* define a constant unicode string */
hex(0x0a 0a \x0a);           /* define a hexadecimal constant value */
block_begin("block");        /* define the beginning of a block */
block_end("block");          /* define the end of a block */
block_size_b32("block");     /* 32 bits big-endian size of a block */
block_size_l32("block");     /* 32 bits little-endian size of a block */
block_size_b16("block");     /* 16 bits big-endian size of a block */
block_size_l16("block");     /* 16 bits little-endian size of a block */
block_size_8("block");       /* 8 bits size of a block */
block_size_8("block");       /* 8 bits size of a block */
block_size_hex_string("a"); /* hexadecimal string size of a block
block_size_dec_string("b"); /* decimal string size of a block */
block_crc32_b("block");      /* crc32 of a block in big-endian */
block_crc32_l("block");      /* crc32 of a block in little-endian */
send("block");               /* send the block */
recv("block");               /* receive the block */
fuzz_string("dummy");        /* fuzz the string "dummy" */
fuzz_string_uni("dummy");    /* fuzz the unicode string "dummy" */
fuzz_hex(0xff ff \xff);      /* fuzz the hexadecimal value */
```

This functionality is deceivingly simple as with it, it is possible to represent the majority of binary and plain text protocol formats.

The main goal of the Autodafé framework is to reduce the size and complexity of the total fuzzing input space to more efficiently focus on areas of the protocol that are likely to result in the discovery of security vulnerabilities. Calculating the input space, or complexity, for a complete fuzz audit is simple. Consider the following simple Autodafé script:

```
fuzz_string("GET");
string(" /");
fuzz_string("index.html");
string(" HTTP/1.1");
hex(0d 0a);
```

Once launched against a target Web server, this script will first iterate through a number of HTTP verb mutations followed by a number of verb argument mutations. Assume that Autodafé contains a fuzz string substitution library with 500 entries. The total number of test cases required to complete this audit is 500 times the number of variables to fuzz, for a total of 1,000 test cases. In most real-world cases, the substitution library will be at least double that size and there could

[17] http://autodafe.sourceforge.net
[18] http://autodafe.sourceforge.net/docs/autodafe.pdf

be hundreds of variables to fuzz. Autodafé applies an interesting technique named Markers Technique to provide a weight to each fuzz variable. Autodafé defines a marker as data, string or numeric, that is controllable by the user (or fuzzer). The applied weights are used to determine the order in which to process the fuzz variables, focusing on those that are more likely to result in the discovery of security vulnerabilities.

To accomplish this task, Autodafé includes a debugger component named adbg. Debugging technologies have traditionally been used along side fuzzers and even included with specific fuzzing tools such as FileFuzz[19], but Autodafé is the first fuzzing framework to explicitly include a debugger. The debugger component is used by Autodafé to set breakpoints on common dangerous APIs such as strcpy(), which is known as a source of buffer overflows, and fprintf(), which is known as a source of format string vulnerabilities. The fuzzer transmits test cases to both the target and the debugger simultaneously. The debugger then monitors dangerous API calls looking for strings originating from the fuzzer.

Every fuzz variable is considered a marker. The weight for an individual marker detected to pass through a dangerous API is increased. Markers that do not touch dangerous APIs are not fuzzed during the first pass. Markers with heavier weights are given priority and fuzzed first. In the event that the debugger detects an access violation, the fuzzer is automatically informed and the responsible test case is recorded. By prioritizing the fuzz variables and ignoring those that never cross a dangerous API, the total input space can be drastically reduced.

Autodafé includes a couple of additional tools to help quick and efficient fuzzer development. The first tool, PDML2AD, can parse pack data XML (PDML) files exported by Ethereal and Wireshark into the block-based Autodafé language. If your target protocol is among the more than 750 protocols that these popular network sniffers recognize, then the majority of the tedious block-based modeling can be handled automatically. Even in the event that your target protocol is unrecognized, PDML2AD can still provide a few shortcuts, as it will automatically detect plain text fields and generate the appropriate calls to hex(), string(), and so on. The second tool, TXT2AD, is a simple shell script that will convert a text file into an Autodafé script. The third and final tool, ADC, is the Autodafé compiler. ADC is useful when developing complex Autodafé scripts as it can detect common errors such as incorrect function names and unclosed blocks.

Autodafé is a well-thought-out advanced fuzzing framework that expands on the groundwork laid out by SPIKE. Autodafé shares many of the same pros and cons as SPIKE. The most impressive feature of this framework is the debugging component, which stands out among the other frameworks. Again, depending on individual needs, the lack of Microsoft Windows support might immediately disqualify this framework from consideration. Modifications to the framework require recompilation and once again code reuse between fuzz scripts is not as transparent and effortless as it could be.

Custom Fuzzer Case Study: Shockwave Flash

A good representation of the currently available fuzzing frameworks was discussed in the previous section. We encourage you to download the individual framework packages and explore the complete documentation and various examples. No single framework stands out as the best of breed, able to handle any job thrown at it better than the others. Instead, one framework might be more applicable depending on the specific target, goals, timeline, budget, and other factors. Familiarity with a few frameworks will provide a testing team with the flexibility to apply the best tool for any given job.

Although most protocols and file formats can be easily described in at least one of the publicly available fuzzing frameworks, there will be instances where none of them are applicable.

[19] http://www.fuzzing.org

When dealing with specialized software or protocol auditing, the decision to build a new, target-specific fuzzer can yield greater results and save time over the life of the audit. In this section we discuss Adobe's Macromedia Shockwave Flash[20] (SWF) file format as a real-world fuzzing target and provide portions of a custom-built testing solution. Security vulnerabilities in Shockwave Flash are typically high impact as some version of Flash Player is installed on almost every desktop on the planet. In fact, modern Microsoft Windows operating systems ship with a functional version of Flash Player by default. Dissecting every detail of a full-blown SWF audit is well beyond the scope of this paper, so only the most relevant and interesting portions are discussed. For further information, up-to-date results, and code listings, visit the www.fuzzing.org.

A couple of factors make SWF a great candidate for a custom fuzzing solution. First, the structure of the SWF file format is completely documented in an Adobe developer reference titled "Macromedia Flash (SWF) and Flash Video (FLV) File Format Specification."[21] The version of the document referenced for this writing was version 8. Unless you are auditing an open source or in-house developed file format, such thorough documentation is not usually available and might limit your testing to more generic approaches. With this specific case, we will most definitely leverage every detail provided by Adobe and Macromedia. Examining the specification reveals that the SWF file format is largely parsed as a bit stream versus the typical byte-level granularity by which most files and protocols are parsed. As SWFs are typically delivered over the Web, streamlined size is likely the motivation behind the choice to parse at the bit level. This adds a level of complexity for our fuzzer and provides further reason to create a custom solution, as none of the fuzzing frameworks we explored in this paper support bit types.

Modeling SWF Files

The first milestone that must be met to successfully fuzz the SWF file format is for our fuzzer to be able to both mutate and generate test cases. SWF consists of a header followed by a series of tags that resemble the following high-level structure:

```
[Header]
  <magic>
  <version>
  <length>
  <rect>
    [nbits]
    [xmin]
    [xmax]
    [ymin]
    [ymax]
  <framerate>
  <framecount>

[FileAttributes Tag]
[Tag]
  <header>
  <data>
    <datatypes>
    <structs>
    ...
[Tag]
  <header>
  <data>
    <datatypes>
    <structs>
```

[20] http://www.macromedia.com/software/flash/about/
[21] http://www.adobe.com/licensing/developer/

```
    ...
...
[ShowFrame Tag]
[End Tag]
```

This is how the various fields break down:

- magic. Consists of three bytes and for any valid SWF file must be “FWS.”
- version. Consists of a single byte representing the numerical version of Flash that generated the file.
- length. A four-byte value indicating the size of the entire SWF file.
- rect. This field is divided into five components:
    - nbits. This is five bits wide and indicates the length, in bits, of each of the next four fields.
    - xmin, xmax, ymin, and ymax. These fields represent the screen coordinates to which to render the Flash content. Note that these fields do not represent pixels but rather *twips,* a Flash unit of measurement that represents 1/20th of a “logical pixel.”

The rect field is where we can start to see the complexity of this file format. If the value of the nbits field is 3, then the total length of the rect portion of the header is 5 + 3 + 3 + 3 + 3 = 17 bits. If the value of the nbits field is 4 then the total length of the rect portion of the header is 5 + 4 + 4 + 4 + 4 = 21 bits. Representing this structure with one of the available fuzzing frameworks is not a simple task. The final two fields in the header represent the speed at which to play back frames and the number of total frames in the SWF file.

Following the header is a series of tags that define the content and behavior of the Flash file. The first tag, FileAttributes, was introduced in Flash version 8 and is a mandatory tag that serves two purposes. First, the tag specifies whether or not the SWF file contains various properties such as a title and description. This information is not used internally but is instead made available for reference by external processes such as search engines. Second, the tag specifies whether the Flash Player should grant the SWF file either local or network file access. Each SWF file has a variable number of remaining tags that fall into one of two categories: definition and control tags. Definition tags define content such as shapes, buttons, and sound. Control tags define concepts such as placement and motion. The Flash Player processes the tags until it hits a ShowFrame tag, at which point content is rendered to screen. The SWF file is terminated with the mandatory End tag.

There are a number of available tags to choose from, each consisting of a two-byte tag header followed by data. Depending on the length of the data, the tag can be defined as long or short. If the length of the data is less than 63 bits, then the tag is considered short and is defined as: [ten-bit tag id] [six-bit data length] [data]. If the data block is larger, then the tag is considered long and is defined with an additional four-byte field in the header that specifies the length of the data: [ten-bit tag id] [111111] [four-byte data length] [data]. Again, we can see that the complexity of the SWF format does not lend itself well to being modeled within any of the aforementioned fuzzing frameworks.

Things only get worse from here. Each tag has its own set of fields. Some contain raw data, whereas others consist of SWF basic types named structures (struct or structs). Each struct consists of a set of defined fields that can include both raw data and even other structs! It still only gets worse. Groups of fields within both tags and structs may or may not be defined depending on the value of another field within that same tag or struct. Even the explanation can get confusing, so let’s start building from the ground up and then show some examples to better explain everything.

To begin, we define a bit field class in Python for representing arbitrary numeric fields of variable bit lengths. We extend this bit field class to define bytes (chars), words (shorts), dwords (longs), and qwords (doubles) as they can be easily defined as 8-bit, 16-bit, 32-bit, and 64-bit instantiations of the bit field class. These basic data structures are accessible under the package Sulley (not to be confused with the Sulley fuzzing framework discussed in the next section):

```
BIG_ENDIAN    = ">"
LITTLE_ENDIAN = "<"

class bit_field (object):
    def __init__ (self, width, value=0, max_num=None):
        assert(type(value) is int or long)

        self.width   = width
        self.max_num = max_num
        self.value   = value
        self.endian  = LITTLE_ENDIAN
        self.static  = False
        self.s_index = 0

        if self.max_num == None:
            self.max_num = self.to_decimal("1" * width)

    def flatten (self):
        '''
        @rtype:  Raw Bytes
        @return: Raw byte representation
        '''

        # pad the bit stream to the next byte boundary.
        bit_stream = ""

        if self.width % 8 == 0:
            bit_stream += self.to_binary()
        else:
            bit_stream  = "0" * (8 - (self.width % 8))
            bit_stream += self.to_binary()


        flattened = ""

        # convert the bit stream from a string of bits into raw bytes.
        for i in xrange(len(bit_stream) / 8):
            chunk = bit_stream[8*i:8*i+8]
            flattened += struct.pack("B", self.to_decimal(chunk))

        # if necessary, convert the endianess of the raw bytes.
        if self.endian == LITTLE_ENDIAN:
            flattened = list(flattened)
            flattened.reverse()
            flattened = "".join(flattened)

        return flattened

    def to_binary (self, number=None, bit_count=None):
        '''
        @type number:     Integer
        @param number:    (Opt., def=self.value) Number to convert
        @type bit_count:  Integer
        @param bit_count: (Opt., def=self.width) Width of bit string

        @rtype:  String
        @return: Bit string
        '''

        if number == None:
            number = self.value

        if bit_count == None:
            bit_count = self.width

        return "".join(map(lambda x:str((number >> x) & 1), \
                  range(bit_count -1, -1, -1)))

    def to_decimal (self, binary):
        '''
        Convert a binary string into a decimal number and return.
        '''

        return int(binary, 2)

    def randomize (self):
        '''
        Randomize the value of this bitfield.
        '''

        self.value = random.randint(0, self.max_num)

    def smart (self):
        '''
        Step the value of this bitfield through a list of smart values.
        '''

        smart_cases = \
        [
            0,
            self.max_num,
            self.max_num / 2,
            self.max_num / 4,
            # etc...
        ]

        self.value    = smart_cases[self.s_index]
        self.s_index += 1


class byte (bit_field):
    def __init__ (self, value=0, max_num=None):
        if type(value) not in [int, long]:
            value = struct.unpack(endian + "B", value)[0]

        bit_field.__init__(self, 8, value, max_num)


class word (bit_field):
    def __init__ (self, value=0, max_num=None:
        if type(value) not in [int, long]:
            value = struct.unpack(endian + "H", value)[0]

        bit_field.__init__(self, 16, value, max_num)


class dword (bit_field):
    def __init__ (self, value=0, max_num=None):
        if type(value) not in [int, long]:
            value = struct.unpack(endian + "L", value)[0]

        bit_field.__init__(self, 32, value, max_num)


class qword (bit_field):
    def __init__ (self, value=0, max_num=None):
        if type(value) not in [int, long]:
            value = struct.unpack(endian + "Q", value)[0]

        bit_field.__init__(self, 64, value, max_num)


# class aliases
bits   = bit_field
char   = byte
short  = word
long   = dword
double = qword
```

The bit_field base class defines the width of the field (width), the maximum number the field is capable of representing (max_num), the value of the field (value), the bit order of the field (endian), a flag specifying whether or not the field's value can be modified (static), and finally an internally used index (s_index). The bit_field class further defines a number of useful functions:

- The flatten() routine converts and returns a byte-bounded sequence of raw bytes from the field.
- The to_binary() routine can convert a numeric value into a string of bits.
- The to_decimal() routine does the opposite by converting a string of bits into a numeric value.
- The randomize() routine updates the field value to a random value within the fields valid range.
- The smart() routine updates the field value through a sequence of "smart" boundaries, and only an excerpt of these numbers is shown.

When constructing a complex type from the bit_field building block, these last two routines can be called to mutate the generated data structure. The data structure can then be traversed and written to a file by recursively calling the flatten() routines of each individual component.

Using these basic types, we can now begin to define the more simple SWF structs, such as RECT and RGB. The following code excerpts show the definitions of these classes, which inherit from a base class not shown (refer to http://www.fuzzing.org for the definition of the base class):

```
class RECT (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)
        self.fields = \
        [
          ("Nbits", sulley.numbers.bits(5, value=31, static=True)),
          ("Xmin" , sulley.numbers.bits(31)),
          ("Xmax" , sulley.numbers.bits(31)),
          ("Ymin" , sulley.numbers.bits(31)),
          ("Ymax" , sulley.numbers.bits(31)),
        ]

class RGB (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)

        self.fields = \
        [
          ("Red"  , sulley.numbers.byte()),
          ("Green", sulley.numbers.byte()),
          ("Blue" , sulley.numbers.byte()),
        ]
```

This excerpt also shows a simple shortcut that was discovered and agreed on during development. To simplify fuzzing, variable width fields are all defined at their maximum length. To represent structs with dependent fields a new primitive, dependent_bit_field, was derived from the bit_field class as shown here:

```
class dependent_bit_field (sulley.numbers.bit_field):
    def __init__ (self, width, value=0, max_num=None, static=False, \
                  parent=None, dep=None, vals=[]):
        self.parent = parent
        self.dep    = dep
        self.vals   = vals

        sulley.numbers.bit_field.__init__(self, width, value, \
                                          max_num, static)

    def flatten (self):
        # if there is a dependency for flattening (including) this
        # structure, then check it.
        if self.parent:
            #                              VVV - object value
            if self.parent.fields[self.dep][1].value not in self.vals:
                # dependency not met, don't include this object.
                return ""

        return sulley.numbers.bit_field.flatten(self)
```

This extended class specifies a field index and value to examine prior to generating and returning data. If the appropriate field referenced by dep does not contain a value from the list vals then no data is returned. The MATRIX struct demonstrates the usage of this newly defined primitive:

```
class MATRIX (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)
        self.fields = \
        [
          ("HasScale"     , sulley.numbers.bits(1)),

          ("NScaleBits"   , dependent_bits(5, 31, parent=self, \
                            dep=0, vals=[1])),
          ("ScaleX"       , dependent_bits(31, parent=self, \
                            dep=0, vals=[1])),
          ("ScaleY"       , dependent_bits(31, parent=self, \
                            dep=0, vals=[1])),

          ("HasRotate"    , sulley.numbers.bits(1)),

          ("NRotateBits"  , dependent_bits(5, 31, parent=self, \
                            dep=4, vals=[1])),
          ("skew1"        , dependent_bits(31, parent=self, \
                            dep=0, vals=[1])),
          ("skew2"        , dependent_bits(31, parent=self, \
                            dep=0, vals=[1])),

          ("NTranslateBits" , sulley.numbers.bits(5, value=31)),
          ("TranslateX"     , sulley.numbers.bits(31)),
          ("TranslateY"     , sulley.numbers.bits(31)),
        ]
```

From this excerpt, the NScaleBits field within the MATRIX struct is defined as a five-bit-wide field with a default value of 31 that is included in the struct only if the value of the field at index 0 (HasScale) is equal to 1. The ScaleX, ScaleY, skew1, and skew2 fields also depend on the HasScale field. In other words, if HasScale is 1, then those fields are valid. Otherwise, those fields should not be defined. Similarly, the NRotateBits field is dependent on the value of the field at index 4 (HasRotate). At the time of writing, more than 200 SWF structs have been accurately defined in this notation.[22]

With all the necessary primitives and structs, we can now begin to define whole tags. First, a base class is defined from which all tags are derived:

```
class base (structs.base):
    def __init__ (self, parent=None, dep=None, vals=[]):
        self.tag_id = None
        (structs.base).__init__(self, parent, dep, vals)

    def flatten (self):
        bit_stream = structs.base.flatten(self)

        # pad the bit stream to the next byte boundary.
        if len(bit_stream) % 8 != 0:
            bit_stream  = "0" * (8-(len(bit_stream)%8)) + bit_stream

        raw = ""

        # convert the bit stream from a string of bits into raw bytes.
        for i in xrange(len(bit_stream) / 8):
            chunk = bit_stream[8*i:8*i+8]
            raw += pack("B", self.to_decimal(chunk))
```

[22] Again, visit http://www.fuzzing.org for the code.

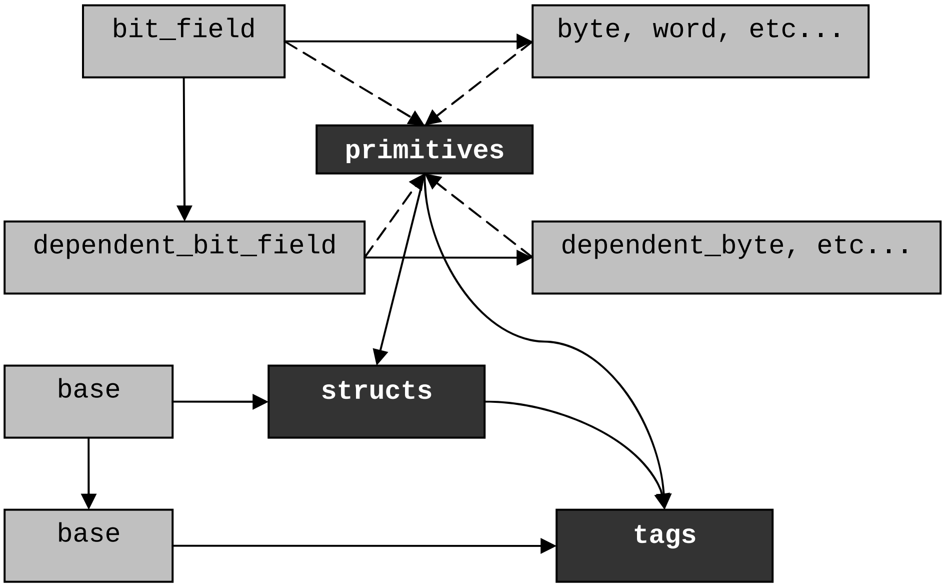
```
        raw_length = len(raw)

        if raw_length >= 63:
            # long (record header is a word + dword)
            record_header = self.tag_id
            record_header <<= 6
            record_header |= 0x3f
            flattened = pack('H', record_header)

            record_header <<= 32
            record_header |= raw_length
            flattened += pack('Q', record_header)
            flattened += raw

        else:
            # short (record_header is a word)
            record_header = self.tag_id
            record_header <<= 6
            record_header |= raw_length
            flattened = pack('H', record_header)
            flattened += raw

        return flattened
```

The flatten() routine for the base tag class automatically calculates the length of the data portion and generates the correct short or long header. At the time of writing more than 50 SWF tags have been accurately defined on this framework. Here are a few examples of simple and medium complexity:

```
class PlaceObject (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)

        self.tag_id = 4
        self.fields = \
        [
          ("CharacterId"    , sulley.numbers.word(value=0x01)),
          ("Depth"          , sulley.numbers.word()),
          ("Matrix"         , structs.MATRIX()),
          ("ColorTransform" , structs.CXFORM()),
        ]


class DefineBitsLossless (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)

        self.tag_id = 20
        self.fields = \
        [
          ("CharacterId"          , sulley.numbers.word()),
          ("BitmapFormat"         , sulley.numbers.byte()),
          ("BitmapWidth"          , sulley.numbers.word()),
          ("BitmapHeight"         , sulley.numbers.word()),
          ("BitmapColorTableSize" , structs.dependent_byte( \
                                    parent=self, dep=1, vals=[3])),
          ("ZlibBitmapData"       , structs.COLORMAPDATA( \
                                    parent=self, dep=1, vals=[3])),
          ("ZlibBitMapData_a"     , structs.BITMAPDATA( \
                                    parent=self, dep=1, vals=[4, 5])),
        ]


class DefineMorphShape (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)

        self.tag_id = 46
        self.fields = \
        [
          ("CharacterId"      , sulley.numbers.word()),
          ("StartBounds"      , structs.RECT()),
          ("EndBounds"        , structs.RECT()),
          ("Offset"           , sulley.numbers.word()),
          ("MorphFillStyles"  , structs.MORPHFILLSTYLE()),
          ("MorphLineStyles"  , structs.MORPHLINESTYLES()),
          ("StartEdges"       , structs.SHAPE()),
          ("EndEdges"         , structs.SHAPE()),
        ]
```

A lot of information has been presented, so let's review. To properly model an arbitrary SWF, we began with the most basic primitive bit_field. From there, we derived primitives for bytes, words, dwords, and qwords. Also from bit_field, we derived dependent_bit_field, from which we further derived dependent bytes, words, dwords, and qwords. These types, in conjunction with a new base class, form the basis for SWF structs. A base class for tags is derived from the structs base class that in conjunction with structs and primitives form SWF tags. The relationship depicted in Figure 1 can further clarify.

Figure 1
SWF fuzzer component relationship



Some other building blocks, such as string primitives, are also necessary for complete SWF modeling. Refer to the source for their definitions. With all these classes in place, we construct a global SWF container and begin instantiating and appending tags to it. The resulting data structure can then be easily traversed and modified either manually or automatically with each individual component's randomize() or smart() routines. Finally, the structure can be written to a file by once again traversing the complex data structure and concatenating the results of the flatten() routine. This design allows for special cases to be transparently handled within individual tags or structs.

Generating Valid Data

Not long after conquering our first challenge of successfully representing basic SWF structures, we come across another. The SWF specification states that tags can only depend on previously defined tags. When fuzzing a tag with multiple dependencies, all of the dependencies must first be successfully parsed; otherwise, the target tag is never reached. Guessing valid field values is a painstaking process, but there is a clever alternative. Utilizing the Google SOAP API,[23] we can search the Web for SWF files with the keyword filetype:swf. A simple script is written to incrementally search for a filetype:swf, b filetype:swf, and through to z filetype:swf. The script parses the returned results looking for and downloading discovered SWF files. Files are named by the MD5 hash to prevent storage of duplicates.

Approximately 10,000 unique SWF files consuming more than 3GB of space have been spidered. Examination of the version field in the SWF header can reveal a statistical sample of the distribution, which is shown in Table 1.

Table 1

Spidered Flash SWF Version Distribution

SWF Version	% of Total
Flash 8	< 1%
Flash 7	~ 2%
Flash 6	~ 11%
Flash 5	~ 55%
Flash 4	~ 28%
Flash 1 - Flash 3	~ 3%

These results are interesting. Unfortunately, no conclusions can be drawn as to the distribution of active Flash Players from these statistics. Another script is then written that parses each file, extracting and storing individual tags. This repository of valid tags can now be used to assist in the creation of SWF test cases.

Fuzzing Environment

The PaiMei[24] reverse engineering framework is used to facilitate the instantiation and monitoring of individual test cases. PaiMei is a set of pure Python classes designed to assist in the development of reverse engineering automation tools. The actual testing process is similar to that used by the generic PaiMei FileFuzz as follows:

1. Load an SWF test case within a Flash Player under the PaiMei PyDbg scripted debugger.

[23] http://www.google.com/apis/index.html

[24] http://openrce.org/downloads/details/208/PaiMei

2. Start the player and monitor execution for five seconds. The delay of five seconds is arbitrarily chosen and assumes that if an error is going to occur during the parsing of the SWF file it will occur within this time frame.
3. If, within that time, an error is detected, store the test case and relevant debug information.
4. If, within that time, no error is detected, close the Flash Player instance.
5. Continue to the next test case.

For more efficient testing, we can modify the frame rate in the SWF header from the default value of 0x000C to the maximum value of 0xFFFF. By doing so, we allow for larger portions of the SWF file to be processed during the five seconds allotted to each test case. The default method of viewing SWF files is to load them in a Web browser such as Microsoft Internet Explorer or Mozilla Firefox. Although this approach is acceptable for this specific case, an alternative method that comes in handy, especially for fuzzing ActionScript,[25] is to use SAFlashPlayer.exe. This small stand-alone player is distributed with Macromedia Studio.[26]

Testing Methodologies

The most generic approach to fuzzing SWF files is to use bit flipping. This rudimentary testing technique has been seen previously at the byte level. Applying this technique at the more granular bit level for SWF is prudent as a single byte might span more than one field. Moving a step beyond bit flipping, a custom-written SWF tag parsing script can be used to change the order of tags within an individual SWF file and swap tags between two different SWFs. Finally, the most thorough approach for SWF fuzzing is to individually stress test every field within every struct and tag. The previously generated tag and struct repository can be referenced to piecemeal a base SWF for mutation.

With all three approaches, the generated files are fed through the fuzzing environment in the same way. In the next section, we explore a new fuzzing framework introduced and released at BlackHat US 2007.

Sulley: Fuzzing Framework

Sulley is a fuzzer development and fuzz testing framework consisting of multiple extensible components. Sulley (in our humble opinion) exceeds the capabilities of most previously published fuzzing technologies, both commercial and those in the public domain. The goal of the framework is to simplify not only data representation, but data transmission and target monitoring as well. Sulley is affectionately named after the creature from Monsters, Inc.[27] because, well, he is fuzzy. You can download the latest version of Sulley from http://www.fuzzing.org/sulley.

Modern-day fuzzers are, for the most part, solely focused on data generation. Sulley not only has impressive data generation, but has taken this a step further and includes many other important aspects a modern fuzzer should provide. Sulley watches the network and methodically maintains records. Sulley instruments and monitors the health of the target, and is capable of reverting to a good state using multiple methods. Sulley detects, tracks, and categorizes detected faults. Sulley can fuzz in parallel, significantly increasing test speed. Sulley can automatically determine what unique sequence of test cases triggers faults. Sulley does all this and more, automatically, and without attendance. Overall usage of Sulley breaks down to the following:

[25] http://en.wikipedia.org/wiki/ActionScript
[26] http://www.adobe.com/products/studio/
[27] http://www.pixar.com/featurefilms/inc/chars_pop1.html

1. *Data representation.* This is the first step in using any fuzzer. Run your target and tickle some interfaces while snagging the packets. Break down the protocol into individual requests and represent them as blocks in Sulley.
2. *Session.* Link your developed requests together to form a session, attach the various available Sulley monitoring agents (socket, debugger, etc.), and commence fuzzing.
3. *Postmortem.* Review the generated data and monitored results. Replay individual test cases.

Once you have downloaded the latest Sulley package from http://www.fuzzing.org, unpack it to a directory of your choosing. The directory structure is relatively complex, so let's take a look at how everything is organized.

Sulley Directory Structure

There is some rhyme and reason to the Sulley directory structure. Maintaining the directory structure will ensure that everything remains organized while you expand the fuzzer with Legos, requests, and utilities. The following hierarchy outlines what you will need to know about the directory structure:

- *archived_fuzzies*. This is a free-form directory, organized by fuzz target name, to store archived fuzzers and data generated from fuzz sessions.
    - *trend_server_protect_5168*. This retired fuzz is referenced during the step-by-step walk-through later in this document.
    - *trillian_jabber.* Another retired fuzz referenced from the documentation.
- *audits*. Recorded PCAPs, crash bins, code coverage, and analysis graphs for active fuzz sessions should be saved to this directory. Once retired, recorded data should be moved to archived_fuzzies.
- *docs*. This is documentation and generated Epydoc API references.
- *requests*. Library of Sulley requests. Each target should get its own file, which can be used to store multiple requests.
    - *__REQUESTS__.html*. This file contains the descriptions for stored request categories and lists individual types. Maintain alphabetical order.
    - *http.py*. Various Web server fuzzing requests.
    - *trend.py*. Contains the requests associated with the complete fuzz walkthrough discussed later in this document.
- *sulley.* The fuzzer framework. Unless you want to extend the framework, you shouldn't need to touch these files.
    - *legos*. User-defined complex primitives.
        - *ber.py*. ASN.1/BER primitives.
        - *dcerpc.py*. Microsoft RPC NDR primitives.
        - *misc.py*. Various uncategorized complex primitives such as e-mail addresses and hostnames.
        - *xdr.py*. XDR types.
    - *pgraph.* Python graph abstraction library. Utilized in building sessions.
    - *utils.* Various helper routines.
        - *dcerpc.py*. Microsoft RPC helper routines such as for binding to an interface and generating a request.
        - *misc.py*. Various uncategorized routines such as CRC-16 and UUID manipulation routines.
        - *scada.py*. SCADA-specific helper routines including a DNP3

block encoder.

    - *__init__.py*. The various s_ aliases that are used in creating requests are defined here.
    - *blocks.py*. Blocks and block helpers are defined here.
    - *pedrpc.py*. This file defines client and server classes that are used by Sulley for communications between the various agents and the main fuzzer.
    - *primitives.py*. The various fuzzer primitives including static, random, strings, and integers are defined here.
    - *sessions.py*. Functionality for building and executing a session.
    - *sex.py*. Sulley's custom exception handling class.
- *unit_tests.* Sulley's unit testing harness.
- *utils.* Various stand-alone utilities.
    - *crashbin_explorer.py*. Command-line utility for exploring the results stored in serialized crash bin files.
    - *pcap_cleaner.py*. Command-line utility for cleaning out a PCAP directory of all entries not associated with a fault.
- *network_monitor.py*. PedRPC-driven network monitoring agent.
- *process_monitor.py*. PedRPC-driven debugger-based target monitoring agent.
- *unit_test.py*. Sulley's unit testing harness.
- *vmcontrol.py.* PedRPC-driven VMWare controlling agent.

Now that the directory structure is a bit more familiar, let's take a look at how Sulley handles data representation. This is the first step in constructing a fuzzer.

Data Representation

Aitel had it right with SPIKE: We've taken a good look at every fuzzer we can get our hands on and the block-based approach to protocol representation stands above the others, combining both simplicity and the flexibility to represent most protocols. Sulley utilizes a block-based approach to generate individual requests, which are then later tied together to form a session. To begin, initialize with a new name for your request:

```
s_initialize("new request")
```

Now you start adding primitives, blocks, and nested blocks to the request. Each primitive can be individually rendered and mutated. Rendering a primitive returns its contents in raw data format. Mutating a primitive transforms its internal contents. The concepts of rendering and mutating are abstracted from fuzzer developers for the most part, so don't worry about it. Know, however, that each mutatable primitive accepts a default value that is restored when the fuzzable values are exhausted.

Static and Random Primitives

Let's begin with the simplest primitive, s_static(), which adds a static unmutating value of arbitrary length to the request. There are various aliases sprinkled throughout Sulley for your convenience, s_dunno(), s_raw(), and s_unknown() are aliases of s_static():

```
# these are all equivalent:
s_static("pedram\x00was\x01here\x02")
s_raw("pedram\x00was\x01here\x02")
s_dunno("pedram\x00was\x01here\x02")
```

```
s_unknown("pedram\x00was\x01here\x02")
```

Primitives, blocks, and so on all take an optional name keyword argument. Specifying a name allows you to access the named item directly from the request via request.names["name"] instead of having to walk the block structure to reach the desired element. Related to the previous, but not equivalent, is the s_binary() primitive, which accepts binary data represented in multiple formats. SPIKE users will recognize this API, as its functionality is (or rather should be) equivalent to what you are already familiar with:

```
    # yeah, it can handle all these formats.
    s_binary("0xde 0xad be ef \xca fe 00 01 02 0xba0xdd f0 0d")
```

Most of Sulley's primitives are driven by fuzz heuristics and therefore have a limited number of mutations. An exception to this is the s_random() primitive, which can be utilized to generate random data of varying lengths. This primitive takes two mandatory arguments, 'min_length' and 'max_length', specifying the minimum and maximum length of random data to generate on each iteration, respectively. This primitive also accepts the following optional keyword arguments:

- *num_mutations (integer, default=25).* Number of mutations to make before reverting to default.
- *fuzzable (boolean, default=True).* Enable or disable fuzzing of this primitive.
- *name (string, default=None).* As with all Sulley objects, specifying a name gives you direct access to this primitive throughout the request.

The num_mutations keyword argument specifies how many times this primitive should be rerendered before it is considered exhausted. To fill a static sized field with random data, set the values for 'min_length' and 'max_length' to be the same.

Integers

Binary and ASCII protocols alike have various-sized integers sprinkled all throughout them, for instance the Content-Length field in HTTP. Like most fuzzing frameworks, a portion of Sulley is dedicated to representing these types:

- one byte: s_byte(), s_char()
- two bytes: s_word(), s_short()
- four bytes: s_dword(), s_long(), s_int()
- eight bytes: s_qword(), s_double()

The integer types each accept at least a single parameter, the default integer value. Additionally the following optional keyword arguments can be specified:

- *endian (character, default='<').* Endianess of the bit field. Specify < for little endian and > for big endian.
- *format (string, default="binary").* Output format, "binary" or "ascii", controls the format in which the integer primitives render. For example, the value 100 is rendered as "100" in ASCII and "\x64" in binary.
- *signed (boolean, default=False).* Make size signed versus unsigned, applicable only when format="ascii".
- *full_range (boolean, default=False).* If enabled, this primitive mutates through all possible values (more on this later).
- *fuzzable (boolean, default=True).* Enable or disable fuzzing of this primitive.
- *name (string, default=None).* As with all Sulley objects specifying a name gives you direct access to this primitive throughout the request.

The *full_range* modifier is of particular interest among these. Consider you want to fuzz a DWORD value; that's 4,294,967,295 total possible values. At a rate of 10 test cases per second, it would take 13 years to finish fuzzing this single primitive! To reduce this vast input space, Sulley defaults to trying only "smart" values. This includes the plus and minus 10 border cases around 0, the maximum integer value (MAX_VAL), MAX_VAL divided by 2, MAX_VAL divided by 3, MAX_VAL divided by 4, MAX_VAL divided by 8, MAX_VAL divided by 16, and MAX_VAL divided by 32. Exhausting this reduced input space of 141 test cases requires only seconds.

Strings and Delimiters

Strings can be found everywhere. E-mail addresses, hostnames, usernames, passwords, and more are all examples of string components you will no doubt come across when fuzzing. Sulley provides the s_string() primitive for representing these fields. The primitive takes a single mandatory argument specifying the default, valid value for the primitive. The following additional keyword arguments can be specified:

- *size (integer, default=-1).* Static size for this string. For dynamic sizing, leave this as -1.
- *padding (character, default='\x00').* If an explicit size is specified and the generated string is smaller than that size, use this value to pad the field up to size.
- *encoding (string, default="ascii").* Encoding to use for string. Valid options include whatever the Python str.encode() routine can accept. For Microsoft Unicode strings, specify "utf_16_le".
- *fuzzable (boolean, default=True).* Enable or disable fuzzing of this primitive.
- *name (string, default=None).* As with all Sulley objects, specifying a name gives you direct access to this primitive throughout the request.

Strings are frequently parsed into subfields through the use of delimiters. The space character, for example, is used as a delimiter in the HTTP request GET /index.html HTTP/1.0. The front slash (/) and dot (.) characters in that same request are also delimiters. When defining a protocol in Sulley, be sure to represent delimiters using the s_delim() primitive. As with other primitives, the first argument is mandatory and used to specify the default value. Also as with other primitives, s_delim() accepts the optional 'fuzzable' and 'name' keyword arguments. Delimiter mutations include repetition, substitution, and exclusion. As a complete example, consider the following sequence of primitives for fuzzing the HTML body tag.

```
# fuzzes the string: <BODY bgcolor="black">
s_delim("<")
s_string("BODY")
s_delim(" ")
s_string("bgcolor")
s_delim("=")
s_delim("\"")
s_string("black")
s_delim("\"")
s_delim(">")
```

Blocks

Having mastered primitives, let's next take a look at how they can be organized and nested within blocks. New blocks are defined and opened with s_block_start() and closed with s_block_end(). Each block must be given a name, specified as the first argument to s_block_start(). This routine also accepts the following optional keyword arguments:

- *group (string, default=None).* Name of group to associate this block with (more on this later).

- *encoder (function pointer, default=None).* Pointer to a function to pass rendered data to prior to returning it.
- *dep (string, default=None).* Optional primitive whose specific value on which this block is dependent.
- *dep_value (mixed, default=None).* Value that field dep must contain for block to be rendered.
- *dep_values (list of mixed types, default=[]).* Values that field dep can contain for block to be rendered.
- *dep_compare (string, default="==").* Comparison method to apply to dependency. Valid options include: ==, !=, >, >=, <, and <=.

Grouping, encoding, and dependencies are powerful features not seen in most other frameworks and they deserve further dissection.

Groups

Grouping allows you to tie a block to a group primitive to specify that the block should cycle through all possible mutations for each value within the group. The group primitive is useful, for example, for representing a list of valid opcodes or verbs with similar argument structures. The primitive s_group() defines a group and accepts two mandatory arguments. The first specifies the name of the group and the second specifies the list of possible raw values to iterate through. As a simple example, consider the following complete Sulley request designed to fuzz a Web server:

```
# import all of Sulley's functionality.
from sulley import *

# this request is for fuzzing: {GET,HEAD,POST,TRACE} /index.html
HTTP/1.1

# define a new block named "HTTP BASIC".
s_initialize("HTTP BASIC")

# define a group primitive listing the various HTTP verbs we wish to
fuzz.
s_group("verbs", values=["GET", "HEAD", "POST", "TRACE"])

# define a new block named "body" and associate with the above group.
if s_block_start("body", group="verbs"):
    # break the remainder of the HTTP request into individual
primitives.
    s_delim(" ")
    s_delim("/")
    s_string("index.html")
    s_delim(" ")
    s_string("HTTP")
    s_delim("/")
    s_string("1")
    s_delim(".")
    s_string("1")
    # end the request with the mandatory static sequence.
    s_static("\r\n\r\n")
# close the open block, the name argument is optional here.
s_block_end("body")
```

The script begins by importing all of Sulley's components. Next a new request is initialized and given the name HTTP BASIC. This name can later be referenced for accessing this request directly. Next, a group is defined with the name verbs and the possible string values GET, HEAD, POST, and TRACE. A new block is started with the name body and tied to the previously defined group primitive through the optional group keyword argument. Note that s_block_start() always returns True, which allows you to optionally “tab out” its contained primitives using a simple if clause. Also note that the name argument to s_block_end() is optional. These framework design decisions were made purely for aesthetic purposes. A series of basic delimiter and string primitives are then defined within the confinements of the body block and the block is closed. When this defined request is loaded into a Sulley session, the fuzzer will generate and transmit all possible values for the block body, once for each verb defined in the group.

Encoders

Encoders are a simple, yet powerful block modifier. A function can be specified and attached to a block to modify the rendered contents of that block prior to return and transmission over the wire. This is best explained with a real-world example. The DcsProcessor.exe daemon from Trend Micro Control Manager listens on TCP port 20901 and expects to receive data formatted with a proprietary XOR encoding routine. Through reverse engineering of the decoder, the following XOR encoding routine was developed:

```
def trend_xor_encode (str):
    key = 0xA8534344
    ret = ""

    # pad to 4 byte boundary.
    pad = 4 - (len(str) % 4)

    if pad == 4:
        pad = 0

    str += "\x00" * pad

    while str:
        dword  = struct.unpack("<L", str[:4])[0]
        str    = str[4:]
        dword ^= key
        ret   += struct.pack("<L", dword)
        key    = dword

    return ret
```

Sulley encoders take a single parameter, the data to encode, and return the encoded data. This defined encoder can now be attached to a block containing fuzzable primitives, allowing the fuzzer developer to continue as if this little hurdle never existed.

Dependencies

Dependencies allow you to apply a conditional to the rendering of an entire block. This is accomplished by first linking a block to a primitive on which it will be dependent using the optional dep keyword parameter. When the time comes for Sulley to render the dependent block, it will check the value of the linked primitive and behave accordingly. A dependent value can be specified with the dep_value keyword parameter. Alternatively, a list of dependent values can be specified with the dep_values keyword parameter. Finally, the actual conditional comparison can be modified through the dep_compare keyword parameter. For example, consider a situation where depending on the value of an integer, different data is expected:

```
s_short("opcode", full_range=True)

# opcode 10 expects an authentication sequence.
if s_block_start("auth", dep="opcode", dep_value=10):
    s_string("USER")
    s_delim(" ")
    s_string("pedram")
    s_static("\r\n")
    s_string("PASS")
    s_delim(" ")
    s_delim("fuzzywuzzy")
s_block_end()

# opcodes 15 and 16 expect a single string hostname.
if s_block_start("hostname", dep="opcode", dep_values=[15, 16]):
    s_string("pedram.openrce.org")
s_block_end()

# the rest of the opcodes take a string prefixed with two underscores.
if s_block_start("something", dep="opcode", dep_values=[10, 15, 16],
dep_compare="!="):
    s_static("__")
    s_string("some string")
s_block_end()
```

Block dependencies can be chained together in any number of ways, allowing for powerful (and unfortunately complex) combinations.

Block Helpers

An important aspect of data generation that you must become familiar with to effectively utilize Sulley is the block helper. This category includes sizers, checksums, and repeaters.

Sizers

SPIKE users will be familiar with the s_sizer() (or s_size()) block helper. This helper takes the block name to measure the size of as the first parameter and accepts the following additional keyword arguments:

- *length (integer, default=4).* Length of size field.
- *endian (character, default='<').* Endianess of the bit field. Specify '<' for little endian and '>' for big endian.
- *format (string, default="binary").* Output format, "binary" or "ascii", controls the format in which the integer primitives render.
- *inclusive (boolean, default=False).* Should the sizer count its own length?
- *signed (boolean, default=False).* Make size signed versus unsigned, applicable only when format="ascii".
- *fuzzable (boolean, default=False).* Enable or disable fuzzing of this primitive.
- *name (string, default=None).* As with all Sulley objects, specifying a name gives you direct access to this primitive throughout the request.

Sizers are a crucial component in data generation that allow for the representation of complex protocols such as XDR notation, ASN.1, and so on. Sulley will dynamically calculate the length of the associated block when rendering the sizer. By default, Sulley will not fuzz size fields. In many cases this is the desired behavior; in the event it isn’t, however, enable the fuzzable flag.

Checksums

Similar to sizers, the s_checksum() helper takes the block name to calculate the checksum of as the first parameter. The following optional keyword arguments can also be specified:

- *algorithm (string or function pointer, default="crc32")*. Checksum algorithm to apply to target block (crc32, adler32, md5, sha1).
- *endian (character, default='<').* Endianess of the bit field. Specify '<' for little

endian and '>' for big endian.

- *length (integer, default=0).* Length of checksum, leave as 0 to autocalculate.
- *name (string, default=None).* As with all Sulley objects, specifying a name gives you direct access to this primitive throughout the request.

The algorithm argument can be one of crc32, adler32, md5, or sha1. Alternatively, you can specify a function pointer for this parameter to apply a custom checksum algorithm.

Repeaters

The s_repeat() (or s_repeater()) helper is used for replicating a block a variable number of times. This is useful, for example, when testing for overflows during the parsing of tables with multiple elements. This helper takes three mandatory arguments: the name of the block to be repeated, the minimum number of repetitions, and the maximum number of repetitions. Additionally, the following optional keyword arguments are available:

- *step (integer, default=1).* Step count between min and max reps.
- *fuzzable (boolean, default=False).* Enable or disable fuzzing of this primitive.
- *name (string, default=None).* As with all Sulley objects, specifying a name gives you direct access to this primitive throughout the request.

Consider the following example that ties all three of the introduced helpers together. We are fuzzing a portion of a protocol that contains a table of strings. Each entry in the table consists of a two-byte string type field, a two-byte length field, a string field, and finally a CRC-32 checksum field that is calculated over the string field. We don't know what the valid values for the type field are, so we'll fuzz that with random data. Here is what this portion of the protocol might look like in Sulley:

```
# table entry: [type][len][string][checksum]
if s_block_start("table entry"):
    # we don't know what the valid types are, so we'll fill this in
with random data.
    s_random("\x00\x00", 2, 2)

    # next, we insert a sizer of length 2 for the string field to
follow.
    s_size("string field", length=2)

    # block helpers only apply to blocks, so encapsulate the string
primitive in one.
    if s_block_start("string field"):
        # the default string will simply be a short sequence of Cs.
        s_string("C" * 10)
    s_block_end()

    # append the CRC-32 checksum of the string to the table entry.
    s_checksum("string field")
s_block_end()

# repeat the table entry from 100 to 1,000 reps stepping 50 elements on
each iteration.
s_repeat("table entry", min_reps=100, max_reps=1000, step=50)
```

This Sulley script will fuzz not only table entry parsing, but might discover a fault in the processing of overly long tables.

Legos

Sulley utilizes legos for representing user-defined components such as e-mail addresses, hostnames, and protocol primitives used in Microsoft RPC, XDR, ASN.1, and others. In ASN.1 / BER strings are represented as the sequence [0x04][0x84][dword length][string]. When fuzzing an ASN.1-based protocol, including the length and type prefixes in front of every string can become cumbersome. Instead we can define a lego and reference it:

```
s_lego("ber_string", "anonymous")
```

Every lego follows a similar format with the exception of the optional options keyword argument, which is specific to individual legos. As a simple example, consider the definition of the tag lego, helpful when fuzzing XMLish protocols:

```
class tag (blocks.block):
    def __init__ (self, name, request, value, options={}):
        blocks.block.__init__(self, name, request, None, None, None,
None)

        self.value   = value
        self.options = options

        if not self.value:
            raise sex.error("MISSING LEGO.tag DEFAULT VALUE")

        #
        # [delim][string][delim]

        self.push(primitives.delim("<"))
        self.push(primitives.string(self.value))
        self.push(primitives.delim(">"))
```

This example lego simply accepts the desired tag as a string and encapsulates it within the appropriate delimiters. It does so by extending the block class and manually adding the tag delimiters and user-supplied string to the block stack via self.push().

Here is another example that produces a simple lego for representing ASN.1 / BER[28] integers in Sulley. The lowest common denominator was chosen to represent all integers as four-byte integers that follow the form: [0x02][0x04][dword], where 0x02 specifies integer type, 0x04 specifies the integer is four bytes long, and the dword represents the actual integer we are passing. Here is what the definition looks like from sulley\legos\ber.py:

```
class integer (blocks.block):
    def __init__ (self, name, request, value, options={}):
        blocks.block.__init__(self, name, request, None, None, None,
None)

        self.value   = value
        self.options = options

        if not self.value:
            raise sex.error("MISSING LEGO.ber_integer DEFAULT VALUE")

        self.push(primitives.dword(self.value, endian=">"))
```

[28] http://luca.ntop.org/Teaching/Appunti/asn1.html

```
    def render (self):
        # let the parent do the initial render.
        blocks.block.render(self)

        self.rendered = "\x02\x04" + self.rendered
        return self.rendered
```

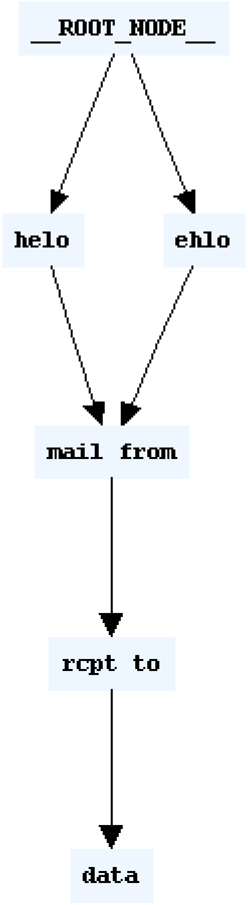
Similar to the previous example, the supplied integer is added to the block stack with self.push(). Unlike the previous example, the render() routine is overloaded to prefix the rendered contents with the static sequence \x02\x04 to satisfy the integer representation requirements previously described. Sulley grows with the creation of every new fuzzer. Developed blocks and requests expand the request library and can be easily referenced and used in the construction of future fuzzers. Now it's time to take a look at building a session.

Session

Once you have defined a number of requests it's time to tie them together in a session. One of the major benefits of Sulley over other fuzzing frameworks is its capability of fuzzing deep within a protocol. This is accomplished by linking requests together in a graph. In the following example, a sequence of requests are tied together and the pgraph library, which the session and request classes extend from, is leveraged to render the graph in uDraw format as shown in Figure 2:

```
from sulley import *

s_initialize("helo")
s_static("helo")

s_initialize("ehlo")
s_static("ehlo")

s_initialize("mail from")
s_static("mail from")

s_initialize("rcpt to")
s_static("rcpt to")

s_initialize("data")
s_static("data")

sess = sessions.session()
sess.connect(s_get("helo"))
sess.connect(s_get("ehlo"))
sess.connect(s_get("helo"),      s_get("mail from"))
sess.connect(s_get("ehlo"),      s_get("mail from"))
sess.connect(s_get("mail from"), s_get("rcpt to"))
sess.connect(s_get("rcpt to"),   s_get("data"))

fh = open("session_test.udg", "w+")
fh.write(sess.render_graph_udraw())
fh.close()
```

Figure 2
Example SMTP session graph structure



When it comes time to fuzz, Sulley walks the graph structure, starting with the root node and fuzzing each component along the way. In this example it begins with the helo request. Once complete, Sulley will begin fuzzing the mail from request. It does so by prefixing each test case with a valid helo request. Next, Sulley moves on to fuzzing the rcpt to request. Again, this is accomplished by prefixing each test case with a valid helo and mail from request. The process continues through data and then restarts down the ehlo path. The ability to break a protocol into individual requests and fuzz all possible paths through the constructed protocol graph is powerful. Consider, for example, an issue disclosed against Ipswitch Collaboration Suite in September 2006.[29] The software fault in this case was a stack overflow during the parsing of long strings contained within the characters @ and :. What makes this case interesting is that this vulnerability is only exposed over the EHLO route and not the HELO route. If our fuzzer is unable to walk all possible protocol paths, then issues such as this might be missed.

When instantiating a session, the following optional keyword arguments can be specified:

- *session_filename (string, default=None).* Filename to which to serialize persistent data. Specifying a filename allows you to stop and resume the fuzzer.
- *skip (integer, default=0).* Number of test cases to skip.
- *sleep_time (float, default=1.0).* Time to sleep in between transmission of test cases.
- *log_level (integer, default=2).* Set the log level; a higher number indicates more log messages.
- *proto (string, default="tcp").* Communication protocol.

[29] http://www.zerodayinitiative.com/advisories/ZDI-06-028.html

- *timeout (float, default=5.0).* Seconds to wait for a send() or recv() to return prior to timing out.

Another advanced feature that Sulley introduces is the ability to register callbacks on every edge defined within the protocol graph structure. This allows us to register a function to call between node transmissions to implement functionality such as challenge response systems. The callback method must follow this prototype:

```
    def callback(node, edge, last_recv, sock)
```

Here, node is the node about to be sent, edge is the last edge along the current fuzz path to node, last_recv contains the data returned from the last socket transmission, and sock is the live socket. A callback is also useful in situations where, for example, the size of the next pack is specified in the first packet. As another example, if you need to fill in the dynamic IP address of the target, register a callback that snags the IP from sock.getpeername()[0]. Edge callbacks can also be registered through the optional keyword argument callback to the session.connect() method.

Targets and Agents

The next step is to define targets, link them with agents, and add the targets to the session. In the following example, we instantiate a new target that is running inside a VMWare virtual machine and link it to three agents:

```
target = sessions.target("10.0.0.1", 5168)

target.netmon    = pedrpc.client("10.0.0.1",  26001)
target.procmon   = pedrpc.client("10.0.0.1",  26002)
target.vmcontrol = pedrpc.client("127.0.0.1", 26003)

target.procmon_options = \
{
    "proc_name"      : "SpntSvc.exe",
    "stop_commands"  : ['net stop "trend serverprotect"'],
    "start_commands" : ['net start "trend serverprotect"'],
}

sess.add_target(target)
sess.fuzz()
```

The instantiated target is bound on TCP port 5168 on the host 10.0.0.1. A network monitor agent is running on the target system, listening by default on port 26001. The network monitor will record all socket communications to individual PCAP files labeled by test case number. The process monitor agent is also running on the target system, listening by default on port 26002. This agent accepts additional arguments specifying the process name to attach to, the command to stop the target process, and the command to start the target process. Finally the VMWare control agent is running on the local system, listening by default on port 26003. The target is added to the session and fuzzing begins. Sulley is capable of fuzzing multiple targets, each with a unique set of linked agents. This allows you to save time by splitting the total test space across the various targets.

Let’s take a closer look at each individual agent’s functionality.

Agent: Network Monitor (network_monitor.py)

The network monitor agent is responsible for monitoring network communications and logging them to PCAP files on disk. The agent is hard-coded to bind to TCP port 26001 and accepts connections from the Sulley session over the PedRPC custom binary protocol. Prior to transmitting a test case to the target, Sulley contacts this agent and requests that it begin recording network traffic. Once the test case has been successfully transmitted, Sulley again contacts this agent, requesting it to flush recorded traffic to a PCAP file on disk. The PCAP files are named by test case number for easy retrieval. This agent does not have to be launched on the same system as the target software. It must, however, have visibility into sent and received network traffic. This agent accepts the following command-line arguments:

```
ERR> USAGE: network_monitor.py
    <-d|--device DEVICE #>    device to sniff on (see list below)
    [-f|--filter PCAP FILTER] BPF filter string
    [-p|--log_path PATH]      log directory to store pcaps to
    [-l|--log_level LEVEL]    log level (default 1), increase for more
verbosity

Network Device List:
    [0] \Device\NPF_GenericDialupAdapter
    [1] {2D938150-427D-445F-93D6-A913B4EA20C0}  192.168.181.1
    [2] {9AF9AAEC-C362-4642-9A3F-0768CDA60942}  0.0.0.0
    [3] {9ADCDA98-A452-4956-9408-0968ACC1F482}  192.168.81.193
    ...
```

Agent: Process Monitor (process_monitor.py)

The process monitor agent is responsible for detecting faults that might occur in the target process during fuzz testing. The agent is hard-coded to bind to TCP port 26002 and accepts connections from the Sulley session over the PedRPC custom binary protocol. After successfully transmitting each individual test case to the target, Sulley contacts this agent to determine if a fault was triggered. If so, high-level information regarding the nature of the fault is transmitted back to the Sulley session for display through the internal Web server (more on this later). Triggered faults are also logged in a serialized “crash bin” for postmortem analysis. This functionality is explored in further detail later. This agent accepts the following command-line arguments:

```
ERR> USAGE: process_monitor.py
    <-c|--crash_bin FILENAME> filename to serialize crash bin class to
    [-p|--proc_name NAME]     process name to search for and attach to
    [-i|--ignore_pid PID]     ignore this PID when searching for the
target process
    [-l|--log_level LEVEL]    log level (default 1), increase for more
verbosity
```

Agent: VMWare Control (vmcontrol.py)

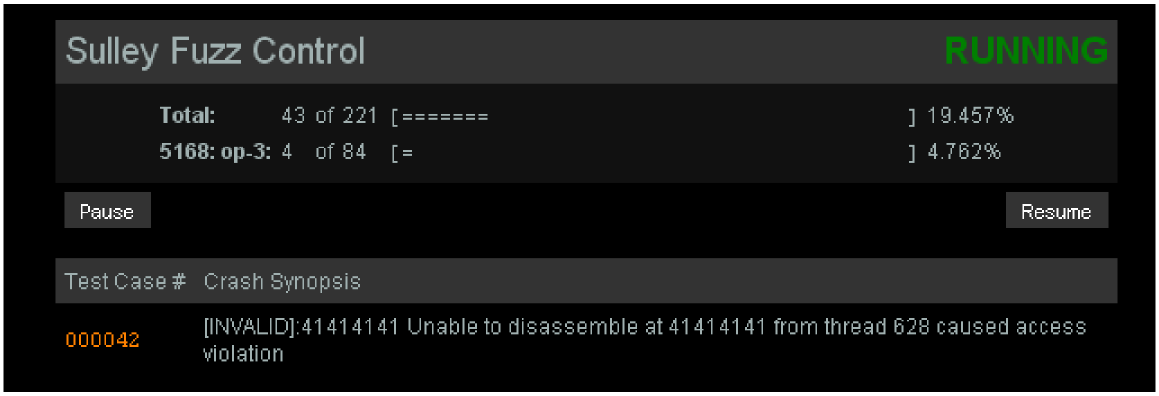
The VMWare control agent is hard-coded to bind to TCP port 26003 and accepts connections from the Sulley session over the PedRPC custom binary protocol. This agent exposes an API for interacting with a virtual machine image, including the ability to start, stop, suspend, or reset the image as well as take, delete, and restore snapshots. In the event that a fault has been detected or the target cannot be reached, Sulley can contact this agent and revert the virtual machine to a known good state. The test sequence honing tool will rely heavily on this agent to accomplish its task of identifying the exact sequence of test cases that trigger any given complex fault. This agent accepts the following command-line arguments:

```
ERR> USAGE: vmcontrol.py
    <-x|--vmx FILENAME>    path to VMX to control
    <-r|--vmrun FILENAME>  path to vmrun.exe
    [-s|--snapshot NAME>   set the snapshot name
    [-l|--log_level LEVEL] log level (default 1), increase for more
verbosity
```

Web Monitoring Interface

The Sulley session class has a built-in minimal Web server that is hard-coded to bind to port 26000. Once the fuzz() method of the session class is called, the Web server thread spins off and the progress of the fuzzer including intermediary results can be seen. An example screen shot is shown in Figure 3.

Figure 3
Sulley Web monitoring interface



```
Total:        43 of 221 [=======                     ] 19.457%
5168: op-3: 4   of 84   [=                           ] 4.762%
```

Test Case #	Crash Synopsis
000042	[INVALID]:41414141 Unable to disassemble at 41414141 from thread 628 caused access violation

The fuzzer can be paused and resumed by clicking the appropriate buttons. A synopsis of each detected fault is displayed as a list with the offending test case number listed in the first column. Clicking the test case number loads a detailed crash dump at the time of the fault. This information is of course also available in the crash bin file and accessible programmatically. Once the session is complete, it's time to enter the postmortem phase and analyze the results.

Postmortem

Once a Sulley fuzz session is complete, it is time to review the results and enter the postmortem phase. The session's built-in Web server will provide you with early indications on potentially uncovered issues, but this is the time you will actually separate out the results. A couple of utilities exist to help you along in this process. The first is the crashbin_explorer.py utility, which accepts the following command-line arguments:

```
$ ./utils/crashbin_explorer.py
    USAGE: crashbin_explorer.py <xxx.crashbin>
        [-t|--test #]     dump the crash synopsis for a specific test
case number
        [-g|--graph name] generate a graph of all crash paths, save to
'name'.udg
```

We can use this utility, for example, to view every location at which a fault was detected and furthermore list the individual test case numbers that triggered a fault at that address. The following results are from a real-world audit against the Trillian Jabber protocol parser:

```
$ ./utils/crashbin_explorer.py audits/trillian_jabber.crashbin
    [3] ntdll.dll:7c910f29 mov ecx,[ecx] from thread 664 caused access
violation
            1415, 1416, 1417,
    [2] ntdll.dll:7c910e03 mov [edx],eax from thread 664 caused access
violation
```

```
            3780, 9215,
    [24] rendezvous.dll:4900c4f1 rep movsd from thread 664 caused
access violation
            1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 3443, 3781,
3782, 3783, 3784, 3785, 3786, 3787, 9216, 9217, 9218, 9219, 9220, 9221,
9222, 9223,
    [1] ntdll.dll:7c911639 mov cl,[eax+0x5] from thread 664 caused
access violation
            3442,
```

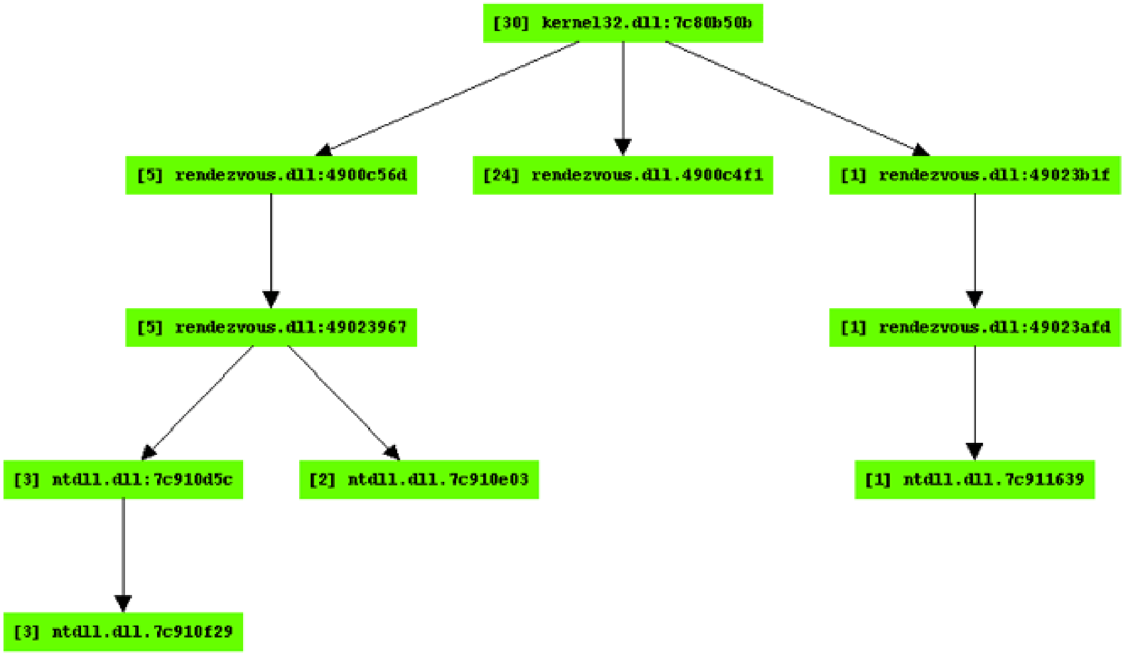
None of these listed fault points might stand out as an obviously exploitable issue. We can drill further down into the specifics of an individual fault by specifying a test case number with the -t command-line switch. Let’s take a look at test case number 1416:

```
$ ./utils/crashbin_explorer.py audits/trillian_jabber.crashbin -t 1416
    ntdll.dll:7c910f29 mov ecx,[ecx] from thread 664 caused access
violation
    when attempting to read from 0x263b7467
    CONTEXT DUMP
      EIP: 7c910f29 mov ecx,[ecx]
      EAX: 039a0318 (  60424984) ->
gt;&gt;&gt;...&gt;&gt;&gt;&gt;&gt;(heap)
      EBX: 02f40000 (  49545216) ->    PP@ (heap)
      ECX: 263b7467 ( 641430631) -> N/A
      EDX: 263b7467 ( 641430631) -> N/A
      EDI: 0399fed0 (  60423888) ->
#e<root><message>&gt;&gt;&gt;&gt;...&gt;&gt;&gt;& (heap)
      ESI: 039a0310 (  60424976) ->
gt;&gt;&gt;...&gt;&gt;&gt;&gt;&gt;(heap)
      EBP: 03989c38 (  60333112) -> \|gt;&t]IP"Ix;IXIox@ @x@PP8|p|Hg9I
P (stack)
      ESP: 03989c2c (  60333100) -> \|gt;&t]IP"Ix;IXIox@ @x@PP8|p|Hg9I
(stack)
      +00: 02f40000 (  49545216) ->    PP@ (heap)
      +04: 0399fed0 (  60423888) ->
#e<root><message>&gt;&gt;&gt;&gt;...&gt;&gt;&gt;& (heap)
      +08: 00000000 (         0) -> N/A
      +0c: 03989d0c (  60333324) -> Hg9I Pt]I@"ImI,IIpHsoIPnIX{ (stack)
      +10: 7c910d5c (2089880924) -> N/A
      +14: 02f40000 (  49545216) ->    PP@ (heap)
    disasm around:
            0x7c910f18 jnz 0x7c910fb0
            0x7c910f1e mov ecx,[esi+0xc]
            0x7c910f21 lea eax,[esi+0x8]
            0x7c910f24 mov edx,[eax]
            0x7c910f26 mov [ebp+0xc],ecx
            0x7c910f29 mov ecx,[ecx]
            0x7c910f2b cmp ecx,[edx+0x4]
            0x7c910f2e mov [ebp+0x14],edx
            0x7c910f31 jnz 0x7c911f21

    stack unwind:
            ntdll.dll:7c910d5c
            rendezvous.dll:49023967
            rendezvous.dll:4900c56d
            kernel32.dll:7c80b50b
    SEH unwind:
            03989d38 -> ntdll.dll:7c90ee18
            0398ffdc -> rendezvous.dll:49025d74
            ffffffff -> kernel32.dll:7c8399f3
```

Again, nothing too obvious might stand out, but we know that we are influencing this specific access violation as the register being invalidly dereferenced, ECX, contains the ASCII string: "&;tg". String expansion issue perhaps? We can view the crash locations graphically, which adds an extra dimension displaying the known execution paths using the -g command-line switch. The following generated graph is again from a real-world audit against the Trillian Jabber parser:

Figure 4

Sulley crash bin graphical exploration


[30] kernel32.dll:7c80b50b
[5] rendezvous.dll:4900c56d
[24] rendezvous.dll.4900c4f1
[1] rendezvous.dll:49023b1f
[5] rendezvous.dll:49023967
[1] rendezvous.dll:49023afd
[3] ntdll.dll:7c910d5c
[2] ntdll.dll.7c910e03
[1] ntdll.dll.7c911639
[3] ntdll.dll.7c910f29

We can see that although we've uncovered four different crash locations, the source of the issue appears to be the same. Further research reveals that this is indeed correct. The specific flaw exists in the Rendezvous/Extensible Messaging and Presence Protocol (XMPP) messaging subsystem. Trillian locates nearby users through the _presence mDNS (multicast DNS) service on UDP port 5353. Once a user is registered through mDNS, messaging is accomplished via XMPP over TCP port 5298. Within plugins\rendezvous.dll, the following logic is applied to received messages:

```
    4900C470 str_len:
    4900C470     mov cl, [eax]      ; *eax = message+1
    4900C472     inc eax
    4900C473     test cl, cl
    4900C475     jnz short str_len

    4900C477     sub eax, edx
    4900C479     add eax, 128       ; strlen(message+1) + 128
    4900C47E     push eax
    4900C47F     call _malloc
```

The string length of the supplied message is calculated and a heap buffer in the amount of length + 128 is allocated to store a copy of the message, which is then passed through expatxml.xmlComposeString(), a function called with the following prototype:

```
    plugin_send(MYGUID, "xmlComposeString", struct xml_string_t *);

    struct xml_string_t {
        unsigned int      struct_size;
        char              *string_buffer;
        struct xml_tree_t *xml_tree;
    };
```

The xmlComposeString() routine calls through to expatxml.19002420(), which, among other things, HTML encodes the characters &, >, and < as &, >, and <, respectively. This behavior can be seen in the following disassembly snippet:

```
    19002492 push 0
    19002494 push 0
    19002496 push offset str_Amp       ; "&"
    1900249B push offset ampersand     ; "&"
    190024A0 push eax
    190024A1 call sub_190023A0

    190024A6 push 0
    190024A8 push 0
    190024AA push offset str_Lt        ; "<"
    190024AF push offset less_than     ; "<"
    190024B4 push eax
    190024B5 call sub_190023A0

    190024BA push
    190024BC push
    190024BE push offset str_Gt        ; ">"
    190024C3 push offset greater_than  ; ">"
    190024C8 push eax
    190024C9 call sub_190023A0
```

As the originally calculated string length does not account for this string expansion, the following subsequent in-line memory copy operation within rendezvous.dll can trigger an exploitable memory corruption:

```
    4900C4EC mov ecx, eax
    4900C4EE shr ecx, 2
    4900C4F1 rep movsd
    4900C4F3 mov ecx, eax
    4900C4F5 and ecx, 3
    4900C4F8 rep movsb
```

Each of the faults detected by Sulley were in response to this logic error. Tracking fault locations and paths allowed us to quickly postulate that a single source was responsible. A final step we might wish to take is to remove all PCAP files that do not contain information regarding a fault. The pcap_cleaner.py utility was written for exactly this task:

```
$ ./utils/pcap_cleaner.py
    USAGE: pcap_cleaner.py <xxx.crashbin> <path to pcaps>
```

This utility will open the specified crash bin file, read in the list of test case numbers that triggered a fault, and erase all other PCAP files from the specified directory. To better understand how everything ties together, from start to finish, we will walk through a complete real-world example audit.

A Complete Walkthrough

This example touches on many intermediate to advanced Sulley concepts and should hopefully solidify your understanding of the framework. Many details regarding the specifics of the target are skipped in this walkthrough, as the main purpose of this section is to demonstrate the usage of a number of advanced Sulley features. The chosen target is Trend Micro Server Protect, specifically a Microsoft DCE/RPC endpoint on TCP port 5168 bound to by the service SpntSvc.exe. The RPC endpoint is exposed from TmRpcSrv.dll with the following Interface Definition Language (IDL) stub information:

```
    // opcode: 0x00, address: 0x65741030
    // uuid: 25288888-bd5b-11d1-9d53-0080c83a5c2c
    // version: 1.0

    error_status_t rpc_opnum_0 (
    [in] handle_t arg_1,                          // not sent on wire
    [in] long trend_req_num,
    [in][size_is(arg_4)] byte some_string[],
    [in] long arg_4,
    [out][size_is(arg_6)] byte arg_5[],           // not sent on wire
    [in] long arg_6
    );
```

Neither of the parameters arg_1 and arg_6 is actually transmitted across the wire. This is an important fact to consider later when we write the actual fuzz requests. Further examination reveals that the parameter trend_req_num has special meaning. The upper and lower halves of this parameter control a pair of jump tables that expose a plethora of reachable subroutines through this single RPC function. Reverse engineering the jump tables reveals the following combinations:

- When the value for the upper half is 0x0001, 1 through 21 are valid lower half values.
- When the value for the upper half is 0x0002, 1 through 18 are valid lower half values.
- When the value for the upper half is 0x0003, 1 through 84 are valid lower half values.
- When the value for the upper half is 0x0005, 1 through 24 are valid lower half values.
- When the value for the upper half is 0x000A, 1 through 48 are valid lower half values.
- When the value for the upper half is 0x001F, 1 through 24 are valid lower half values.

We must next create a custom encoder routine that will be responsible for encapsulating defined blocks as a valid DCE/RPC request. There is only a single function number, so this is simple. We define a basic wrapper around utisl.dcerpc.request(), which hard-codes the opcode parameter to zero:

```
# dce rpc request encoder used for trend server protect 5168 RPC
service.
```

```
# opnum is always zero.
def rpc_request_encoder (data):
    return utils.dcerpc.request(0, data)
```

Building the Requests

Armed with this information and our encoder we can begin to define our Sulley requests. We create a file requests\trend.py to contain all our Trend-related request and helper definitions and begin coding. This is an excellent example of how building a fuzzer request within a language (as opposed to a custom language) is beneficial as we take advantage of some Python looping to automatically generate a separate request for each valid upper value from trend_req_num:

```
for op, submax in [(0x1, 22), (0x2, 19), (0x3, 85), (0x5, 25), (0xa,
49), (0x1f, 25)]:
    s_initialize("5168: op-%x" % op)
    if s_block_start("everything", encoder=rpc_request_encoder):
        # [in] long trend_req_num,
        s_group("subs", values=map(chr, range(1, submax)))
        s_static("\x00")                  # subs is actually a little
endian word
        s_static(struct.pack("<H", op))  # opcode

        # [in][size_is(arg_4)] byte some_string[],
        s_size("some_string")
        if s_block_start("some_string", group="subs"):
            s_static("A" * 0x5000, name="arg3")
        s_block_end()

        # [in] long arg_4,
        s_size("some_string")

        # [in] long arg_6
        s_static(struct.pack("<L", 0x5000)) # output buffer size
    s_block_end()
```

Within each generated request a new block is initialized and passed to our previously defined custom encoder. Next, the s_group() primitive is used to define a sequence named subs that represents the lower half value of trend_req_num we saw earlier. The upper half word value is next added to the request stream as a static value. We will not be fuzzing the trend_req_num as we have reverse engineered its valid values; had we not, we could enable fuzzing for these fields as well. Next, the NDR size prefix for some_string is added to the request. We could optionally use the Sulley DCE/RPC NDR lego primitives here, but because the RPC request is so simple we decide to represent the NDR format manually. Next, the some_string value is added to the request. The string value is encapsulated in a block so that its length can be measured. In this case we use a static-sized string of the character A (roughly 20k worth). Normally we would insert an s_string() primitive here, but because we know Trend will crash with any long string, we reduce the test set by utilizing a static value. The length of the string is appended to the request again to fulfill the size_is requirement for arg_4. Finally, we specify an arbitrary static size for the output buffer size and close the block. Our requests are now ready and we can move on to creating a session.

Creating the Session

We create a new file in the top-level Sulley folder named fuzz_trend_server_protect_5168.py for our session. This file has since been moved to the archived_fuzzies folder because it has completed its life. First things first, we import Sulley and the created Trend requests from the request library:

```
from sulley   import *
from requests import trend
```

Next, we are going to define a presend function that is responsible for establishing the DCE/RPC connection prior to the transmission of any individual test case. The presend routine accepts a single parameter, the socket on which to transmit data. This is a simple routine to write thanks to the availability of utils.dcerpc.bind(), a Sulley utility routine:

```
def rpc_bind (sock):
    bind = utils.dcerpc.bind("25288888-bd5b-11d1-9d53-0080c83a5c2c",
"1.0")
    sock.send(bind)

    utils.dcerpc.bind_ack(sock.recv(1000))
```

Now it's time to initiate the session and define a target. We'll fuzz a single target, an installation of Trend Server Protect housed inside a VMWare virtual machine with the address 10.0.0.1. We'll follow the framework guidelines by saving the serialized session information to the audits directory. Finally, we register a network monitor, process monitor, and virtual machine control agent with the defined target:

```
sess   =
sessions.session(session_filename="audits/trend_server_protect_5168.ses
sion")
target = sessions.target("10.0.0.1", 5168)

target.netmon    = pedrpc.client("10.0.0.1",  26001)
target.procmon   = pedrpc.client("10.0.0.1",  26002)
target.vmcontrol = pedrpc.client("127.0.0.1", 26003)
```

Because a VMWare control agent is present, Sulley will default to reverting to a known good snapshot whenever a fault is detected or the target is unable to be reached. If a VMWare control agent is not available but a process monitor agent is, then Sulley attempts to restart the target process to resume fuzzing. This is accomplished by specifying the stop_commands and start_commands options to the process monitor agent:

```
target.procmon_options = \
{
    "proc_name"      : "SpntSvc.exe",
    "stop_commands"  : ['net stop "trend serverprotect"'],
    "start_commands" : ['net start "trend serverprotect"'],
}
```

The proc_name parameter is mandatory whenever you use the process monitor agent; it specifies what process name to which the debugger should attach and in which to look for faults. If neither a VMWare control agent nor a process monitor agent is available, then Sulley has no choice but to simply provide the target time to recover in the event a data transmission is unsuccessful.

Next, we instruct the target to start by calling the VMWare control agents restart_target() routine. Once running, the target is added to the session, the presend routine is defined, and each of the defined requests is connected to the root fuzzing node. Finally, fuzzing commences with a call to the session classes' fuzz() routine.

```
# start up the target.
```

```
target.vmcontrol.restart_target()

print "virtual machine up and running"

sess.add_target(target)
sess.pre_send = rpc_bind
sess.connect(s_get("5168: op-1"))
sess.connect(s_get("5168: op-2"))
sess.connect(s_get("5168: op-3"))
sess.connect(s_get("5168: op-5"))
sess.connect(s_get("5168: op-a"))
sess.connect(s_get("5168: op-1f"))
sess.fuzz()
```

Setting Up the Environment

The final step before launching the fuzz session is to set up the environment. We do so by bringing up the target virtual machine image and launching the network and process monitor agents directly within the test image with the following command-line parameters:

```
network_monitor.py -d 1 \
                   -f "src or dst port 5168" \
                   -p audits\trend_server_protect_5168

process_monitor.py -c audits\trend_server_protect_5168.crashbin \
                   -p SpntSvc.exe
```

Both agents are executed from a mapped share that corresponds with the Sulley top-level directory from which the session script is running. A Berkeley Packet Filter (BPF) filter string is passed to the network monitor to ensure that only the packets we are interested in are recorded. A directory within the audits folder is also chosen where the network monitor will create PCAPs for every test case. With both agents and the target process running, a live snapshot is made as named sulley ready and waiting.

Next, we shut down VMWare and launch the VMWare control agent on the host system (the fuzzing system). This agent requires the path to the vmrun.exe executable, the path to the actual image to control, and finally the name of the snapshot to revert to in the event of a fault discovery of data transmission failure:

```
vmcontrol.py -r "c:\Progra~1\VMware\VMware~1\vmrun.exe" \
             -x "v:\vmfarm\images\windows\2000\win_2000_pro-
clones\TrendM~1\win_2000_pro.vmx" \
             --snapshot "sulley ready and waiting"
```

Ready, Set, Action! And Postmortem

Finally, we are ready. Simply launch fuzz_trend_server_protect_5168.py, connect a Web browser to http://127.0.0.1:26000 to monitor the fuzzer progress, sit back, watch, and enjoy.

When the fuzzer completes running through its list of 221 test cases, we discover that 19 of them triggered faults. Using the crashbin_explorer.py utility we can explore the faults categorized by exception address:

```
$ ./utils/crashbin_explorer.py
audits/trend_server_protect_5168.crashbin
    [6] [INVALID]:41414141 Unable to disassemble at 41414141 from
thread 568 caused access violation
            42, 109, 156, 164, 170, 198,
    [3] LogMaster.dll:63272106 push ebx from thread 568 caused access
violation
            53, 56, 151,
    [1] ntdll.dll:77fbb267 push dword [ebp+0xc] from thread 568 caused
access violation
            195,
    [1] Eng50.dll:6118954e rep movsd from thread 568 caused access
violation
            181,
    [1] ntdll.dll:77facbbd push edi from thread 568 caused access
violation
            118,
    [1] Eng50.dll:61187671 cmp word [eax],0x3b from thread 568 caused
access violation
            116,
    [1] [INVALID]:0058002e Unable to disassemble at 0058002e from
thread 568 caused access violation
            70,
    [2] Eng50.dll:611896d1 rep movsd from thread 568 caused access
violation
            152, 182,
    [1] StRpcSrv.dll:6567603c push esi from thread 568 caused access
violation
            106,
    [1] KERNEL32.dll:7c57993a cmp ax,[edi] from thread 568 caused
access violation
            165,
    [1] Eng50.dll:61182415 mov edx,[edi+0x20c] from thread 568 caused
access violation
            50,
```

Some of these are clearly exploitable issues, for example, the test cases that resulted with an EIP of 0x41414141. Test case 70 seems to have stumbled on a possible code execution issue as well, a Unicode overflow (actually this can be a straight overflow with a bit more research). The crash bin explorer utility can generate a graph view of the detected faults as well, drawing paths based on observed stack backtraces. This can help pinpoint the root cause of certain issues. The utility accepts the following command-line arguments:

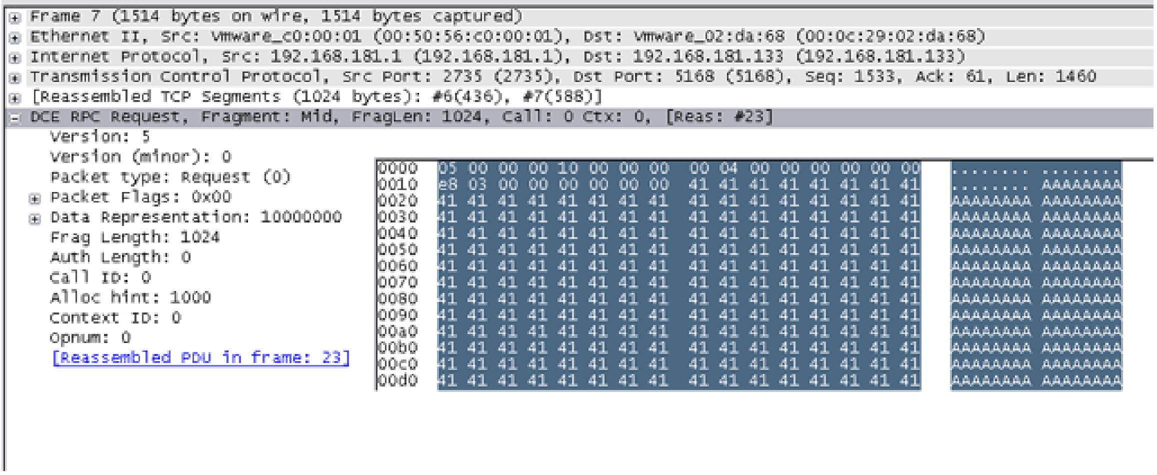
```
$ ./utils/crashbin_explorer.py
    USAGE: crashbin_explorer.py <xxx.crashbin>
        [-t|--test #]     dump the crash synopsis for a specific test
case number
        [-g|--graph name] generate a graph of all crash paths, save to
'name'.udg
```

We can, for example, further examine the CPU state at the time of the fault detected in response to test case 70:

```
$ ./utils/crashbin_explorer.py
audits/trend_server_protect_5168.crashbin -t 70
    [INVALID]:0058002e Unable to disassemble at 0058002e from thread
568 caused access violation
    when attempting to read from 0x0058002e
    CONTEXT DUMP
      EIP: 0058002e Unable to disassemble at 0058002e
      EAX: 00000001 (         1) -> N/A
      EBX: 0259e118 (  39444760) -> A.....AAAAA (stack)
      ECX: 00000000 (         0) -> N/A
      EDX: ffffffff (4294967295) -> N/A
      EDI: 00000000 (         0) -> N/A
      ESI: 0259e33e (  39445310) -> A.....AAAAA (stack)
      EBP: 00000000 (         0) -> N/A
      ESP: 0259d594 (  39441812) -> LA.XLT.......MPT.MSG.OFT.PPS.RT
(stack)
      +00: 0041004c (   4259916) -> N/A
      +04: 0058002e (   5767214) -> N/A
      +08: 0054004c (   5505100) -> N/A
      +0c: 0056002e (   5636142) -> N/A
      +10: 00530042 (   5439554) -> N/A
      +14: 004a002e (   4849710) -> N/A
    disasm around:
            0x0058002e Unable to disassemble
    SEH unwind:
            0259fc58 -> StRpcSrv.dll:656784e3
            0259fd70 -> TmRpcSrv.dll:65741820
            0259fda8 -> TmRpcSrv.dll:65741820
            0259ffdc -> RPCRT4.dll:77d87000
            ffffffff -> KERNEL32.dll:7c5c216c
```

You can see here that the stack has been blown away by what appears to be a Unicode string of file extensions. You can pull up the archived PCAP file for the given test case as well. Figure 5 shows an excerpt of a screen shot from Wireshark examining the contents of one of the captured PCAP files.

Figure 5
Wireshark DCE/RPC dissector



```
⊞ Frame 7 (1514 bytes on wire, 1514 bytes captured)
⊞ Ethernet II, Src: Vmware_c0:00:01 (00:50:56:c0:00:01), Dst: Vmware_02:da:68 (00:0c:29:02:da:68)
⊞ Internet Protocol, Src: 192.168.181.1 (192.168.181.1), Dst: 192.168.181.133 (192.168.181.133)
⊞ Transmission Control Protocol, Src Port: 2735 (2735), Dst Port: 5168 (5168), Seq: 1533, Ack: 61, Len: 1460
⊞ [Reassembled TCP Segments (1024 bytes): #6(436), #7(588)]
⊟ DCE RPC Request, Fragment: Mid, FragLen: 1024, Call: 0 Ctx: 0, [Reas: #23]
    Version: 5
    Version (minor): 0
    Packet type: Request (0)
  ⊞ Packet Flags: 0x00
  ⊞ Data Representation: 10000000
    Frag Length: 1024
    Auth Length: 0
    Call ID: 0
    Alloc hint: 1000
    Context ID: 0
    Opnum: 0
    [Reassembled PDU in frame: 23]

0000  05 00 00 00 10 00 00 00  00 04 00 00 00 00 00 00   ........ ........
0010  e8 03 00 00 00 00 00 00  41 41 41 41 41 41 41 41   ........ AAAAAAAA
0020  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0030  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0040  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0050  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0060  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0070  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0080  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0090  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00a0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00b0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00c0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00d0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
```

A final step we might wish to take is to remove all PCAP files that do not contain information regarding a fault. The pcap_cleaner.py utility was written for exactly this task:

```
$ ./utils/pcap_cleaner.py
    USAGE: pcap_cleaner.py <xxx.crashbin> <path to pcaps>
```

This utility will open the specified crash bin file, read in the list of test case numbers that triggered a fault, and erase all other PCAP files from the specified directory. The discovered code execution vulnerabilities in this fuzz were all reported to Trend and have resulted in the following advisories:

- TSRT-07-01: Trend Micro ServerProtect StCommon.dll Stack Overflow Vulnerabilities
- TSRT-07-02: Trend Micro ServerProtect eng50.dll Stack Overflow Vulnerabilities

This is not to say that all possible vulnerabilities have been exhausted in this interface. In fact, this was the most rudimentary fuzzing possible of this interface. A secondary fuzz that actually uses the s_string() primitive as opposed to simply a long string can now be beneficial.

Summary

Fuzzing frameworks provide a flexible, reusable, and homogenous development environment for bug hunters and QA teams. After going through the list of available fuzzing frameworks in this paper, it should be fairly obvious that there is not a clear winner. All of them provide a reasonable framework to build on, but your selection of a specific framework will generally be driven by the target to be fuzzed and familiarity, or lack thereof, with the programming language used by a particular framework. The Adobe Macromedia Shockwave Flash's SWF file format case study illustrates that frameworks remain an immature technology and you are still likely to encounter situations where none of the existing frameworks are capable of meeting your needs. As demonstrated, it might on occasion thus be necessary to develop a custom fuzzing solution, leveraging reusable components developed for the task at hand. Hopefully this case study was thought provoking and will provide some guidance when you encounter your own exotic fuzzing tasks.

In the final section of this paper we introduced and explored a new fuzzing framework, Sulley. Various aspects of this fuzzing framework were dissected to clearly demonstrate the advancements made by this endeavor. Sulley is the latest in an increasingly long line of fuzzing frameworks and is actively maintained by the author. Check http://www.fuzzing.org for ongoing updates, further documentation, and up-to-date examples.