A Pattern-Matching File Fuzzer for Windows (Revised November 2007)

Lane P. Feltis, Natalie M. Silvanovich, and Neema Teymory

Abstract — **Fuzzing is an emerging area of automated security testing that involves providing semi-random input to a software system. This paper describes a novel fuzzer we designed to overcome the limitations of existing fuzzers. It also details an analysis of this fuzzer's performance.**

Index Terms — **Software Security, Fuzzing, Fuzz Testing, Automatic Test Software**

I. Introduction

FUZZING IS a popular and developing technique for finding implementation vulnerabilities in software. While a large number of fuzzers have been developed over the past few years, few have been designed for fuzzing file-types in the Microsoft Windows environment. This paper describes a fuzzer we designed for this purpose. It also presents the results of experiments we performed, which include the discovery of one serious security vulnerability in Brava! PDF. We also compare our fuzzer to existing technology for finding software vulnerabilities.

II. Fuzzing

A. Defining Fuzzing

Fuzzing is a form of automated software testing that involves providing semi-random input to a software system with the goal of causing system failure. 'Failure' can be defined as desired by those performing the testing, with the limitation that it must be detectable in an automated fashion. By far, the most common application of fuzzing is in software security testing, where it is used to reveal implementation vulnerabilities. It is particularly effective in this context, as it produces the same sort of unintuitive and assumption-free input that typically exposes security vulnerabilities.

B. Fuzzing Challenges

A major challenge in designing a fuzzer is deciding how to generate the semi-random input that is provided to the software under test. On a high level, there are three properties that are desired in this process. Firstly, the generated input should be efficient, meaning that it reveals as many bugs as possible in the software under test in as little time as possible. Secondly, it should be capable of revealing a variety of bugs and not leave portions of the software out when testing (a property called 'coverage'). Finally, the tool should not require extensive configuration to produce input for a certain piece of software. Unfortunately, these properties are at odds with each other, so tradeoffs must be made in fuzzer design [1].

To start, the level of randomness of the input produced by a fuzzer affects both its efficiency and its coverage. Highly random input is almost always rejected by software due to checks on superficial input properties such as header constants and field lengths, so highly random fuzzers are inefficient. Meanwhile, reducing the amount of randomness in a fuzzer reduces the variety of inputs it can create, reducing its coverage. In selecting the level of randomness of a fuzzer, there is a tradeoff between efficiency and coverage.

Another choice in fuzzer design is how much knowledge the fuzzer will have of the format being fuzzed. Of course, this is not always a choice. For example, if a company is fuzzing software they are considering purchasing or a closed-source library they are planning on building into their software, they may not know the input format at all. That said, even if the format is fully known, configuring a fuzzer to work with a specific format is very time consuming. On the other hand, it allows for fuzzing that has complete coverage and is very efficient. This is a second trade-off in fuzzing: a fuzzer can be made more efficient by building in knowledge of the input format being fuzzed, but at the expense of the time it takes to acquire that knowledge and put it into a format that the fuzzer can understand.

C. Current Fuzzing Technology

Currently, a number of fuzzing techniques are used that vary in the randomness of the input they create and the level of knowledge of the format they require.

The most basic fuzzers, brute-force fuzzers, generate purely random input. They are easy to write and require no configuration, but are very inefficient, as they do not make any attempt to conform to even basic input requirements [2].

Mutation fuzzers are a step up from brute-force fuzzers. They take a sample of valid input, called a template, and produce fuzzed input by flipping random bits in the template. The number of bits that are altered can often be specified as a parameter to the fuzzer. An advantage of mutation fuzzers is that they require minimal set-up and no knowledge of the input format other than the template. Unfortunately, superficial input validation in the software being fuzzed often rejects the fuzzed input created by mutation fuzzers [1]. Also, mutation fuzzers

Manuscript received November 19, 2007.
L. P. Feltis (e-mail: lane.feltis@hotmail.com).
N. M. Silvanovich (e-mail: natalies@interchange.ubc.ca).
N. Teymory (e-mail: nteymory@hotmail.com).

may not find certain classes of vulnerabilities, such as buffer overflows, that require an extension, not just an alteration of field values. In addition, any sort of fuzzing involving templates runs the risk of having its coverage severely limited by the template that is selected. This is a very serious problem, as vulnerabilities tend to occur in uncommonly-used (and therefore less-tested) software features. These features are unlikely to be used by the typical input contained in a template, and therefore will not be covered by the fuzzer at all.

Another type of template fuzzing is pattern-matching. Pattern-matching fuzzers go through a template file looking for patterns, such as ASCII strings, and make changes based on these patterns. They are more efficient than mutation fuzzers, but have the same coverage problems, as they also use templates. Pattern-matching fuzzing is used by software companies [3]; however, there are not a lot of tools using this technique that are available to the public.

Block-based fuzzing is a technique that requires a complete knowledge of the input format of the software under test. To use a block-based fuzzer, one creates a script that describes the input format and the fuzzer uses this as a basis for creating corrupt input. Block-based fuzzing has been shown to be very efficient [4], but configuring it for a new input type is time-consuming.

III. Our Fuzzer

We decided to create a file fuzzer for the Microsoft Windows environment, as there are a limited number of tools for this purpose currently available. Many mutation fuzzers have been written, but they are not very efficient and must run for an extended period of time before they find vulnerabilities. Block-based fuzzing technology is also quite mature, but these tools take a lot of time and skill to configure. We wished to create a tool that is more effective than a mutation fuzzer, but does not require the extensive configuration needed by a block-based fuzzer.

We decided to use pattern-matching as our fuzzing technique, as while companies have reported that is has been effective in software testing, few such tools are available to the public (the only one we are aware of is Mistress, which has limited functionality). In addition, we wished to address the limitations of template fuzzing by creating a tool that uses multiple templates.

Our fuzzer has three components: a webcrawler that finds template files, an engine the uses these files to create fuzzed files and a launcher that opens these fuzzed files using the software under test and detects and records failures. The function and design of these elements is described as follows.

A. The Webcrawler

Since limiting the variety of templates used by a fuzzer limits it coverage, it is imperative that the fuzzer uses templates that are representative of the scope of valid input. The ideal template-finding mechanism would have access to the set of all valid input to the software under test and randomly select templates from this set. Templates provided by this mechanism would be completely unbiased towards popular features of the software under test; the number of templates using a particular feature would only be determined by the number of possible inputs that use this feature. Of course, it is not possible to obtain the set of all valid files of a certain type, as the number of such files is infinite, so we attempt to approximate this set by using the set of all files available on the Internet. The goal of our webcrawler is then to select and download random files of a given type from the web.

Our webcrawler uses the Google search engine to do this. To find a single template, the webcrawler selects a random word from a multi-lingual dictionary and queries Google for files of the desire type containing this word. It then randomly selects a template file from those returned.

Unfortunately, this template-selection process is still biased towards files that use popular software features, as these files are more likely to appear on the Internet (this is the definition of popular!). In addition, the fact that all files provided by this tool are reasonably well-ranked on Google may present some hidden biases. That said, all existing file fuzzers use a single template and provide no mechanism or guidance for finding this template, so it is likely our approach provides substantial benefit over what is available.

B. The Engine

Our fuzzing engine takes the template files produced by the webcrawler and uses them to produce fuzzed files. It analyses templates one-at-a-time, and does not produce inputs that combine templates.

To create fuzzed files, our fuzzer identifies ASCII strings within the template file, and resizes them and changes their content. Resizing strings can reveal buffer overflows in software, meanwhile changing the content of strings can reveal format string vulnerabilities and vulnerabilities due to poor error handling. This technique is usually called ‘string extension,’ although strings can also be truncated or remain the same size. We selected this method as it is a staple of block-based fuzzing. Since string-extension has been shown to be effective in finding vulnerabilities in fuzzers that have complete knowledge of the format being fuzzed [4], we suspected that it might also be effective in template fuzzing.

The first step in string-extension is locating strings. Our engine does this by searching the template file for ASCII strings, and making note of their location, length and termination in a string table. Strings that are shorter than five characters, as well as those in the first twenty bytes of the template are not included in the string table as these are more often than not the result of false positives in string identification.

Once all the stings have been identified, the engine goes back and tries to identify each string’s length field. The length field is a construct that is placed before a string and specifies its length. Our fuzzing engine attempts to identify the length field by searching the area of the file before a string for a structure that may represent its length. If one is found, its position and properties (such as length and endianness) are recorded in the string table.

When the string table is complete, it is time to use string-extension to create fuzzed files. To do this, a random string is first selected from the string table. It is then extended using a

randomly-selected string-extension algorithm. Our fuzzer supports three such algorithms: extend-insert, extend-overwrite and extend-rotate. Extend-insert is used in block-based fuzzers, such as SPIKE [4], as well as a number of network protocol fuzzers [5, 6]. Extend-overwrite and extend-rotate are novel and are described below. An example of how each algorithm extends a string is shown in Fig. 1.

1) Extend-Insert

The extend-insert algorithm assigns a new length to the selected string and resizes it to be that length. The new length is selected by a random algorithm that is biased towards boundary values such as 0x7F, 0x80 and 0xFF. The string's length field is then updated to display the new length.

Original File

06	48	75	6D	61	6E	73	03		H	u	m	a	n	s	
61	72	65	09	69	6E	63	61	a	r	e		i	n	c	a
70	61	62	6C	65	02	6F	66	p	a	b	l	e		o	f
07	73	74	6F	72	69	6E	67		s	t	o	r	i	n	g
12	68	69	67	68	2D	71	75		h	i	g	h	-	q	u
61	6C	69	74	79	0D	63	72	a	l	i	t	y		c	r
79	70	74	6F	67	72	61	70	y	p	t	o	g	r	a	p
68	69	63	04	6B	65	79	73	h	i	c		k	e	y	s

Extend-Insert

06	48	75	6D	61	6E	73	03		H	u	m	a	n	s	
61	72	65	09	69	6E	63	61	a	r	e		i	n	c	a
70	61	62	6C	65	02	6F	66	p	a	b	l	e		o	f
07	73	74	6F	72	69	6E	67		s	t	o	r	i	n	g
12	68	69	67	68	2D	71	75		h	i	g	h	-	q	u
42	42	42	42	42	61	6C	69	B	B	B	B	B	A	l	i
74	79	0D	63	72	79	70	74	t	y		c	r	y	p	t
6F	67	72	61	70	68	69	63	o	g	r	a	p	h	i	c
04	6B	65	79	73					k	e	y	s			

Extend-Overwrite

06	48	75	6D	61	6E	73	03		H	u	m	a	n	s	
61	72	65	09	69	6E	63	61	a	r	e		i	n	c	a
70	61	62	6C	65	02	6F	66	p	a	b	l	e		o	f
07	73	74	6F	72	69	6E	67		s	t	o	r	i	n	g
12	68	69	67	68	2D	71	75		h	i	g	h	-	q	u
61	6C	69	74	79	42	42	42	a	l	i	t	y	B	B	B
42	42	74	6F	67	72	61	70	B	B	t	o	g	r	a	p
68	69	63	04	6B	65	79	73	h	i	c		k	e	y	s

Extend-Rotate

06	48	75	6D	61	6E	73	03		H	u	m	a	n	s	
61	72	65	09	69	6E	63	61	a	r	e		i	n	c	a
70	61	62	6C	65	02	6F	66	p	a	b	l	e		o	f
07	73	74	6F	72	69	6E	67		s	t	o	r	i	n	g
12	68	69	67	68	2D	71	75		h	i	g	h	-	q	u
61	6C	69	74	79	42	42	42	a	l	i	t	y	B	B	B
42	42	42	42	42	42	42	42	B	B	B	B	B	B	B	B
42	42	42	42	01	63	01	6B	B	B	B	B		c		k

Fig. 1. String extension algorithms

Bytes are inserted into or deleted from the string to make it the new length. The remaining bytes in the file are shifted to accommodate this change.

In all string-extension algorithms used by our fuzzer, there are three methods that are used to generate the extra bytes that are added to a string (the one used in any particular instance is selected randomly). One of these is pure, non-ASCII random bytes. The second is a series of format strings. The third is the character 'B', resulting in a string with only ASCII characters. The character 'B' is used instead of a random sequence of ASCII characters because it tends to cause Windows to throw an exception in the case of a heap overflow [7].

2) Extend-Overwrite

The extend-overwrite algorithm is identical to extend-insert, except instead of shifting the byes after an extended string to make room for the extra bytes, the new bytes simply overwrite whatever is there. If the new string length is shorter than the selected string length, the length field is updated and a null character is inserted at the correct location for the new length of the string, if the original string was null terminated. Note that this means if the string selected to be extended by extend-overwrite does not have a length field or a null termination and the algorithm selects a new length for this string that is shorter than it was originally, the fuzzed file produced will be identical to the template. Fortunately, this scenario is unlikely, as most strings are null terminated and have length fields, and the length-selection algorithm is much more likely to extend a string than to truncate it.

The rationale behind extend-overwrite is that some file types use absolute offsets in file parsing, and would outright reject files with inserted or deleted bytes (an example of such a file type is the Microsoft Excel format). The extend-overwrite algorithm alters strings without upsetting these offsets. Of course, the segment of the file overwritten by the extra bytes is severely damaged, so it is possible that the file is rejected, even if offsets are not disturbed. The extend-rotate algorithm presents a more elegant solution to this problem.

3) Extend- Rotate

Extend-rotate attempts to extend the selected string in the template file without altering offsets within the file. It does this by shortening adjacent strings in the document, and adding the extra characters to the selected string. The length fields of all strings are updated so they will appear valid. The current implementation reduces the size of the four strings following the selected string to one.

Extend-rotate does not affect the offsets of a file or invalidate large portions of it as extend-overwrite does; however, it is much more adversely affected by errors in the string table than extend-overwrite is.

Errors in the string table occur for two reasons. First, there are false positives in identifying strings. For example, if a byte array happens to randomly have five sequential ASCII characters in it, it will be identified as a string even though it is not really one. Secondly, there are situations where there is uncertainty in determining the length field of a string. Fig. 2 shows an ambiguous string.

The string "M Turing is oft considered the father of modern computer science" has 0x41 characters.

41	4D	20	54	75	72	69	6E
67	20	69	73	20	6F	66	74
20	63	6F	6E	73	69	64	65
72	65	64	20	74	68	65	20
66	61	74	68	65	72	20	6F
66	20	6D	6F	64	65	72	6E
20	63	6F	6D	70	75	74	65
72	20	73	63	69	65	6E	63
65	2E						

A	M		T	u	r	i	n
g		i	s		o	f	t
	c	o	n	s	i	d	e
r	e	d		t	h	e	
f	a	t	h	e	r		o
f		m	o	d	e	r	n
	c	o	m	p	u	t	e
r		s	c	i	e	n	c
e	.						

Fig. 2. Ambiguous string

Unfortunately, 0x41 is also the ASCII representation of the character 'A'. Thus, one cannot be sure whether the 'A' is part of the string or the length field without knowing more about the file format. In this situation, our fuzzer assumes that all ASCII characters are part of the string, but regardless of how this situation is handled, it will result in some strings being misidentified.

Returning to string alteration, clearly a string that is misidentified will not be altered correctly, but since extend-rotate requires a sequence of five strings to all be correctly identified, it is less likely to work correctly than the other alteration methods. For example, if one out of every five strings is misidentified, extend-overwrite will work 80% of the time, whereas extend-rotate will work only 33% of the time. This difference widens as misidentified strings become more common. For this reason, we decided to use both extend-overwrite and extend-rotate for offset-neutral string extension.

4) Compression Support

Since many file formats compress portions of their content using GZIP compression, we added compression support. If the string parser encounters a non-ASCII stream that starts with the correct token and successfully decompresses resulting in a buffer that contains ASCII strings, this compressed string will be noted in the string table. If it is selected to be extended, the stream will be decompressed and parsed, and have one of its strings extended. It will then be recompressed, and the stream in the template file replaced with this new stream, resulting in a fuzzed file.

C. The Effect of Templates

The launcher we produced for our tool is fairly standard. It sequentially launches the software under test with each fuzzed file as a parameter, and then hooks into the process. If it detects an exception, it logs it, otherwise it kills the process. The amount of time the launcher waits before killing an exception-free process can be selected by the user.

IV. Testing The Tool

Since our fuzzer is novel in several respects, there were several questions we wished to answer with regards to its performance. First, we wished to compare the efficiency of our fuzzer to existing template fuzzers. Second, we wanted to compare the effectiveness of the three string-extension algorithms to ensure they are all worthwhile, and to decide how to weight them in selection. Finally, we wanted to explore the effect of using multiple templates on the variety of vulnerabilities discovered.

A. Comparing Our Fuzzer to Mutation Fuzzers

1) Procedure and Results

Mutation fuzzing is currently the most commonly-used template fuzzing technique, so we compared our fuzzer with a popular mutation fuzzer: iDefence's FileFuzz. There is very little variation in the input-generation algorithm used by mutation fuzzers, so the specific tool selected is not particularly important.

We decided to focus on the PDF file format for the purposes of testing, as the file format is publically available but there is no common API available for parsing it. The result of this is that a number of free, independently-implemented PDF readers are available.

We tested the following PDF readers with both our fuzzer and FileFuzz: FoxIt Reader, SamutraPDF, Brava! PDF and Easy PDF 2 Text. We produced 8400 fuzzed files using 20 different templates with each fuzzer, and tested them against each piece of software. The results of these tests are shown in Table 1. From these results, it is evident that our fuzzer performs better than FileFuzz.

Code execution vulnerabilities refer to those that can be exploited to execute arbitrary assembly code. Of the two such vulnerabilities we found, we confirmed that one was exploitable by coding up an exploit that allowed for arbitrary code execution. The second was similar enough that it did not seem necessary to repeat this exercise.

2) Discussion

We were surprised by the large performance improvement our fuzzer provided over FileFuzz; we were expecting a more modest difference. One possible explanation for this is that since mutation fuzzers are widely used, these programs may have already been fuzzed by such a tool and the corresponding vulnerabilities fixed.

B. Evaluating String- Extension Algorithms

Of the 8400 fuzzed files tested against Brava! PDF, three hundred and fifty-nine of them revealed vulnerabilities in the software (twenty-one of which were unique). When these files were generated, the string-extension technique used was recorded. Therefore, it is possible to determine which string-extension techniques created files that revealed vulnerabilities (we will call these successful files). The results of this analysis are shown in Table 2.

TABLE I
Vulnerabilities found by our fuzzer and FileFuzz

		Brava! PDF	Easy PDF 2 Text	Samutra PDF	FoxIt Reader
Our Fuzzer	DOS Vulnerabilities	19	1	3	0
	Code Execution Vulnerabilities	2	0	0	0
FileFuzz	DOS Vulnerabilities	0	1	0	0
	Code Execution Vulnerabilities	0	0	0	0

TABLE II
Extension Algorithms Used to Produce Successful Inputs

Technique	Number of Successful Files	Percentage of Successful Files
Extend-Insert	101	28%
Extend-Overwrite	83	23%
Extend-Rotate	175	49%

This analysis shows that while all three methods produce files that reveal vulnerabilities, extend-rotate is the most effective. In the second version of our fuzzer, we weighted this technique more heavily when selecting a string-extension algorithm to run on a particular string.

C. *The Effect of Templates*

The large number of vulnerabilities revealed in Brava! PDF makes it possible to analyze how template-selection affects the effectiveness of a fuzzer. Fig. 3 shows the number of vulnerabilities found by each template. Different colors represent different unique vulnerabilities. Templates 19 and 20 appear to be more effective than the others, and roughly half of the templates do not reveal vulnerabilities at all. Removing the unique vulnerability that dominates the graph, Fig. 4 shows the other vulnerabilities more clearly.

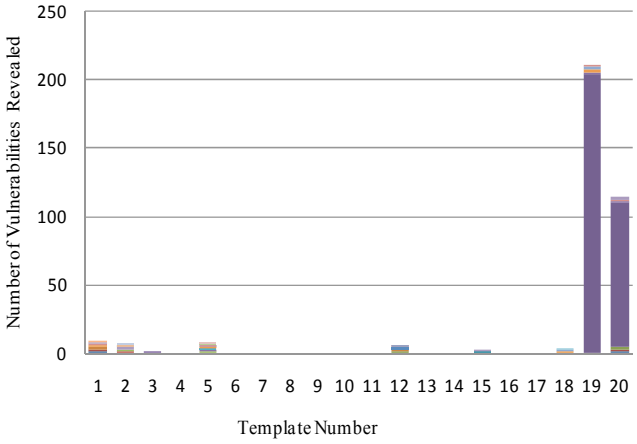

Fig. 3. Vulnerabilities revealed, by template

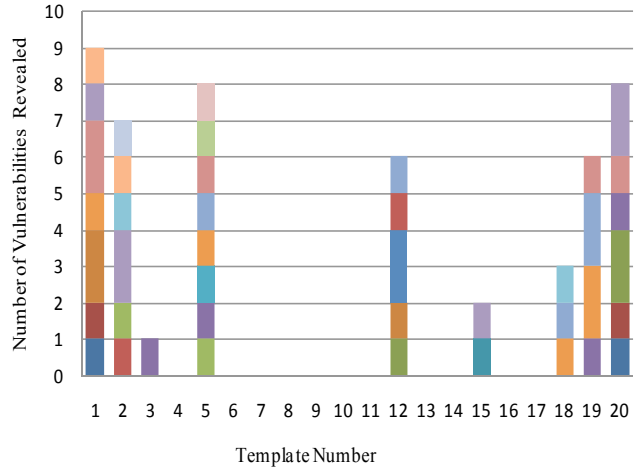

Fig. 4. Vulnerabilities revealed, by template with dominating vulnerability removed

It is interesting to note that while some templates find a good spread of vulnerabilities, no template finds all of them. This suggests that the webcrawler was an important addition to our fuzzer; if a single template was used, it could potentially be one that finds no vulnerabilities. Even if it wasn't, it would not find as many as all the templates combined.

V. Areas For Future Work

Some difficulties encountered in the design process raised some questions that merit future work. We originally intended to make our fuzzer work with binary file types as well as those with ASCII strings; however, it immediately became apparent that ascertaining the position and length field of a byte array is unfeasible. While the problem of ambiguous string occurs in ASCII files, it occurs much more frequently in binary files to the extent that every possible byte-array is ambiguous. Research into how this issue can be resolved would be important to the design of new fuzzers, as well as other fields, such as reverse engineering.

We also only took a very preliminary look at the effect of template selection on fuzzing efficiency, more information on this would be useful. We noted that some templates were more effective than others, so another interesting question is where there is a way to predict which templates will be effective in advance, so that ineffective ones can be avoided.

References

[1] J. DeMott. "The Evolving Art of Fuzzing," presented at DEFCON 14, Las Vegas, Nevada, 2006.

[2] A. Greene, M. Sutton. "The Art of File Format Fuzzing" presented at BlackHat Japan, Tokyo, Japan, 2005.

[3] P. Oelhet. "Violating Assumptions with Fuzzing." *IEEE Security and Privacy,* vol. 3, issue 2, pp. 58-62, March-April 2005.

[4] D. Aitel. "The Advantages of Block-Based Protocol Analysis for Security Testing." Internet: http://www.net-security.org/dl/articles/advantages_of_block_based_analysis.pdf, Feb. 4, 2002 [Nov. 17, 2007].

[5] "Taof – The Art of Fuzzing Using Python." Internet: http://www.secguru.com/link/taof_the_art_of_fuzzing_using_python, Nov. 2006 [Nov. 17, 2007].

[6] "Peach Tutorial." Internet: http://peachfuzz.sourceforge.net/docs/tutorial/peach-tutorial.htm [Nov. 17, 2007].

[7] D. Aitel. "MSRPC Fuzzing with SPIKE 2006." Internet: http://xcon.xfocus.org/xcon2006/archieves/Dave_Aitel-Microsoft_System_RPC_Fuzz.pdf, August 2006 [Nov. 17, 2007]