Analysis of Mutation and Generation-Based Fuzzing

Whitepaper

Charlie Miller and Zachary N. J. Peterson
Independent Security Evaluators
www.securityevaluators.com

March 1, 2007

Abstract

We present a study of two methods of dynamic application analysis: mutation-based fuzzing and generation-based fuzzing. We quantify the differences of these methods by measuring the amount of executed code required to parse PNG image files. Results indicate that generation-based fuzzing can execute 76% more code when compared to mutation-based methods.

1 Introduction

> "Intelligent fuzzing usually gives more results."
> – *Ilja van Sprundel*[8]

Dynamic analysis, or *fuzzing*, is a popular method of finding security vulnerabilities in software [10]. Fuzzing may be used by a developer to find potential problems as part of the quality-assurance process. Likewise, a fuzzer may be used to find potential exploits in an existing software application. The technique of fuzzing consists of sending a large volume of different inputs into a program in an attempt to make the program perform in a manner that was not intended. Fuzzing may result in memory corruption, a program crash, extreme resource usage, *etc*. Such incidents may be exploited to cause a denial of service or even allow an attacker to execute arbitrary code in the context of the application. Fuzzing has grown in popularity because it is much easier (and often more effective) to generate and run arbitrary inputs than it is to perform a manual code audit or use software reverse engineering.

One of the most important aspects of successfully finding vulnerabilities by fuzzing is the quality and quantity of the fuzzed inputs. These inputs, or *test cases*, are normally constructed in one of two fashions. In the first method, called *mutation-based* fuzzing, known good data is collected (files, network traffic, *etc*) and then modified; modifications may be random or heuristic. Examples of heuristic mutations include replacing small strings with longer strings or changing length values to either very large or very small values. The other method, known as *generation-based* fuzzing, starts from a specification or RFC, which describes the file format or network protocol, and constructs test cases from these documents. The key to making effective test cases is to make each case differ from valid data so as to (hopefully) cause a problem in the application, but not to make the data too invalid, or else the target application may quickly discard the input as invalid.

The advantage to mutation-based fuzzing is that little or no knowledge of the protocol or application under study is required. All that is needed is one or more good samples and a method of fuzzing a target application. On the other hand, generation-based fuzzing requires a significant amount of up-front work to study the specification and manually generate test cases. Sometimes manually generated test cases become too similar to the specification and do not differ in the unpredictable ways that benefit generation-based fuzzing. Regardless, intuition says that the extra knowledge gained by understanding the format should result in higher quality test cases. Van Sprundels quote at the beginning of this section summarizes this belief. However, there has not yet been an attempt to quantify how much better generation-based fuzzing performs than mutation-based fuzzing.

This paper takes one specific file format, the Portable Network Graphics (PNG) format, and attempts to precisely quantify the potential advantages gained by using a generation-based approach. It does not attempt to quantify the added difficulties in constructing test cases in a generated form or to establish the number of test cases required to completely fuzz an application. Our results show that generation-based fuzzing performs up to 76% better when compared to mutation-based fuzzing techniques.

2 Portable Network Graphics Files

The Portable Network Graphics (PNG) format is an extensible file format for the loss-less storage of compressed raster images. This format is widely used and is supported by most Internet web browsers including Internet Explorer, Firefox, and Safari.

A PNG file begins with an eight byte signature containing the following values: `137 80 78 71 13 10 26 10`. This signature is followed by a sequence of *chunks*. Each chunk consists of a four byte length field, a four byte chunk type field, an optional chunk data field, and a four byte cyclic redundancy code (CRC) checksum field. The length field is an unsigned integer that gives the length of

Figure 1: The image containing the most chunks (nine).

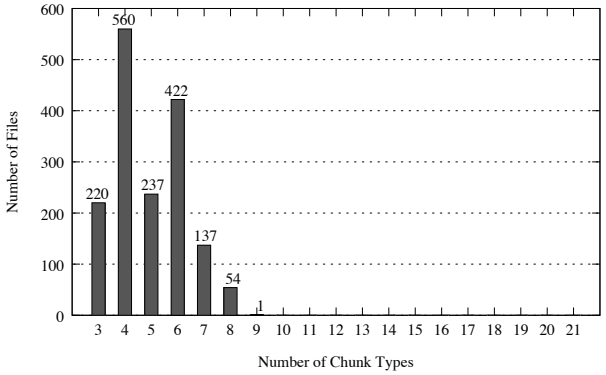

Figure 2: The distribution of the number of chunks present in a file.

the data chunk field in bytes. The chunk type field is designed so that the four bytes are in the ASCII range and the case of the letters has significance for the decoder.

The PNG specification [9] defines eighteen chunk types of which three are mandatory in every PNG file: `IHDR`, `IDAT` and `IEND`. Optional chunks are referred to as *ancillary*. Ancillary chunks may be ignored by a decoder. An extension to the original PNG specification exists [4], from which we consider three additional chunks. In this paper, we consider a total of twenty-one different types of chunks: `IHDR`, `PLTE`, `tRNS`, `cHRM`, `gAMA`, `iCCP`, `IDAT`, `SBIT`, `sRGB`, `tEXt`, `zTXt`, `iTXt`, `bKGD`, `hIST`, `pHYs`, `sPLT`, `tIME`, `oFFs`, `pCAL`, `sCAL`, `IEND`

Some chunk types may occur more than once in a file. Additionally, some chunks are mutually exclusive. For example, if the `iCCP` chunk is present, the `sRGB` chunk should not be present. Furthermore, the ordering of some chunks is mandated. For example, the `IHDR` chunk should be first and the `IEND` chunk should be last. Some chunks, such as the `tIME` chunk, may occur anywhere in the file, while other chunks, such as the `bKGD` chunk, must occur after the `PLTE` chunk but before the `IDAT` chunk. Lastly, the specification also allows other chunks in addition the twenty-one we consider, to be present in a file. We do not consider these for this paper.

3 PNG File Format Statistics

A collection of 1,631 unique PNG files were collected from the Internet. To obtain these files, the Alexa Web Search Platform (AWSP) [1] was used to obtain a list of all URLs ending in ".png". After processing 16,623 potential PNG URLs using tools available through AWSP, 1,631 valid PNG files were obtained. The URLs that did not produce a valid PNG file were either no longer available, were not PNG files, were duplicates of other files obtained, or required authentication to acquire. The valid files were dissected according to the specification and statistics were obtained on the number and types of chunks present in each file (see Table 1).

For each file, we counted the number of chunks that comprised the PNG image. Results are shown in Figure 2. Very few files have more than seven chunks and none have no more than nine chunks. Figure 3 illustrates the frequency of chunk type in the PNG files. As expected, all files contained the mandatory `IHDR`, `IDAT` and `IEND` chunks. However, nine of the twenty-one considered chunk types occurred in fewer than 5% of the files and some chunk types failed to appear at all.

These results demonstrate that choosing random files from the Internet to perform mutation-based fuzzing, without knowledge of the protocol, will generally only fuzz a few different chunk types. On average, only five of the possible twenty-one chunks will be tested by randomly selecting a file – a significant limitation. Vulnerabilities

Number of Files	Mean	Standard Deviation	Maximum	Minimum
1631	4.9	1.3	9	3

Table 1: Distribution of the number of chunks in a file.

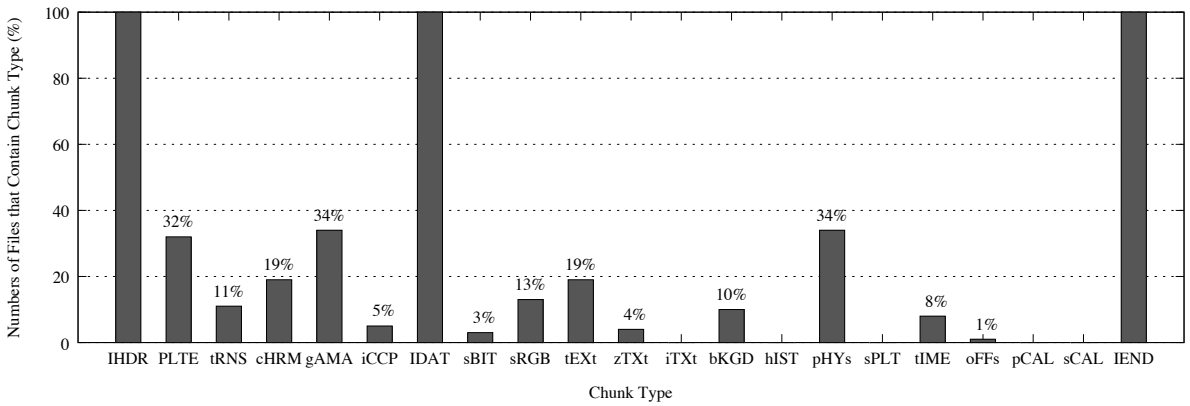

Figure 3: The frequency of type chunk type in files.

cannot be discovered in code that is not executed, exemplifying the advantages of generation-based fuzzing. However, it is not possible to make the conclusion that generation-based fuzzing is more effective without knowing more about the binaries that decode the PNG. For example, perhaps all commodity decoders ignore all but the most common types of chunks, in which case it is only important to fuzz those chunks that are present in common PNGs. We address this concern by correlating code coverage of the PNG decoder with the different chunk types.

4 Analysis of Code Coverage for PNG Chunks

In order to draw a conclusion about the differences between mutation and generation-based fuzzing, it is necessary to see the types of chunks that mutation-based fuzzing is likely to miss, but also to observe the amount of code these chunks represent in the PNG decoder. In other words, missing a chunk that has very little unique processing involved may not be as detrimental as missing a chunk that requires a significant amount of parsing and processing.

Code coverage is a metric used to describe the number lines of source code (or assembly) that have been executed. We use code coverage to measure the amount of code used to process each chunk. While there is not necessarily a correlation between code coverage and finding security vulnerabilities, it is certainly the case that code that is not executed will not reveal any vulnerabilities.

The PNG decoder we choose to examine is a small front-end to libpng 1.2.16 [5]. This library is used by most open source web browsers for PNG decoding, including Firefox, Opera, and Safari. In order to obtain code coverage information, the library was instrumented with gcov [7] to record the source lines executed. As a first step, the coverage for opening an empty file was measured in order to reveal the general processing and start-up code. Next, a minimal PNG file was obtained that contained only the mandatory chunks, namely `IHDR`, `IDAT`, and `IEND`. The difference between the coverage of the minimal file and the coverage opening an empty file is considered to be the minimal amount of code needed to parse a PNG using libpng. We can conclude that this difference excludes gen-

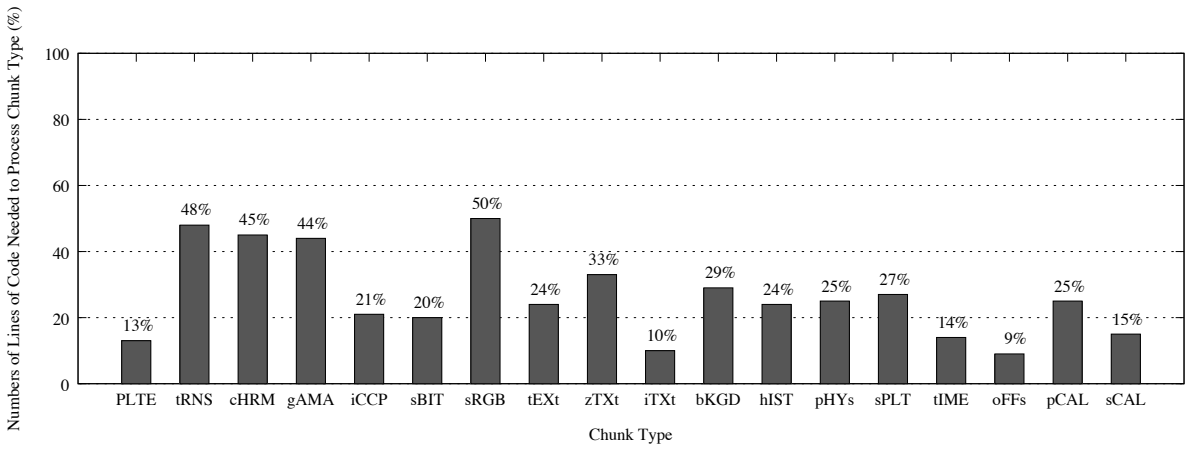

Figure 4: The number of lines of code required to process each chunk as a percentage of the amount of code required to process a minimal PNG file.

eral startup and file I/O and only measures the coverage of the required instructions to decode the three mandatory chunks.

In order to measure the effects of each chunk type on code coverage, we use generation-based fuzzing to incrementally add new chunk types to the minimal PNG file and re-measure code coverage. We used an open-source generation-based fuzzing tool called SPIKEfile [3] to dynamically create approximately 1,000 files for each added chunk type. Chunks were added one at a time, with the only exception being the `hIST` chunk which requires the `PLTE` chunk. We configured SPIKEfile to create PNG file variations that share the same chunk types and structure, but differ in chunk type properties and sizes. This variation provides an insight into how chunk types affect code coverage.

Figure 4 illustrates the approximate increase of code required to parse each chunk type as a percentage of the amount of code required to parse the minimal PNG file. Note that due to the nature of the fuzzing, and the fact that there are often dependencies between different chunks (especially the `IHDR` chunk), these numbers represent a lower bound on the amount of code required for processing each chunk.

Results show that some chunk types require more code than others, but all require a significant amount of code. In particular, chunk types that were not represented at all in the files collected (`iTXt`, `sPLT`, `sCAL`, and `hIST`) together represent 76% more code than is required to process the three mandatory chunks in the minimal PNG file. This code could not have be covered using mutation-based fuzzing. To achieve a maximum code coverage, one must use files that include all the different chunks types. This comes at the cost of a significant initial effort to completely understand the file format.

5 Mutation Versus Generation-based Fuzzing

We formally compare the code coverage of mutation-based and generation-based fuzzing techniques. Two different variations of a mutation-based fuzzer were used. The first takes a known good file and randomly manipulates bytes in various chunk types but does update the corresponding CRC. This represents the most basic type of mutation-based fuzzer as it does not insert new bytes into the file, for example, long strings. The second similarly manipulates random bytes, but generates a correct correspond-

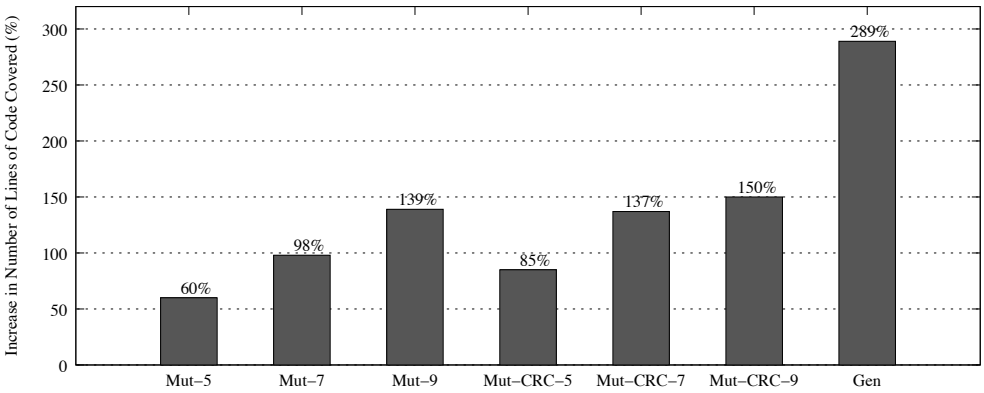

Figure 5: The number of lines of source code covered as a percentage of the amount of code required to decode a minimal PNG file.

ing CRC. This assumes some basic knowledge about the file type. Our mutation-based fuzzer functions very similarly to the mutation-based fuzzers, FILEfuzz [6] and NotSPIKEfile [2].

We ran our mutation-based fuzzer starting from three known good files. The first file contains five different chunk types that would be the most likely number to find by chance, the second contains seven chunk types that would be unlikely to find by chance, and the last contains nine chunk types that would be extremely unlikely to find by chance. For each of these three starting files, 200,000 test cases were generated: 100,000 randomly mutated files, and 100,000 mutated files with matching CRCs.

For generation-based fuzzing, all of the files created for the previous section using SPIKEfile were tested. Additionally, files were created that fuzzed some of the non-data fields, such as block length, CRC, chunk name, *etc*. This resulted in a set of 29,511 test cases that covered all twenty-one chunk types. The set of test cases was significantly smaller than those used by the mutation-based fuzzer and no randomness was used in the generation of the generation-based test cases – they are strictly heuristic. This is a distinct advantage of generation-based fuzzing techniques.

The comparison of code coverage results are shown in Figure 5. "Mut" refers to the mutation-based fuzzing techniques, for the five, seven and nine chunk type test cases, with and without CRCs. "Gen" refers to the generation-based test cases. These results confirm the cursory analysis of the previous sections. The initial file with five chunk types contained the mandatory three chunk types, plus `bKGD` and `pHYs`. Figure 4 estimates that these two chunks add an additional 55% of code coverage over the minimal PNG file. Figure 5 indicates that fuzzing beginning from this initial file covers 60% more code than the minimal file. In this case, Figure 4 serves as a method of estimating code coverage for the actual fuzzing runs.

Despite the clear difference in fuzzing methods, our experimental procedure has limitations. Fuzzing is not an exact science and the results presented are only an indication of a trend. Code coverage could change significantly by varying only a few factors, such as longer run times, more detailed test cases, *etc*. However, due to the fact that applications often contain large sections of code that will only execute with uncommon inputs, mutation-based fuzzers will always fair poorly when compared to generation-based fuzzers. We confirmed this hypothesis with our PNG experiments. For the PNG file format, on average, a mutation-based fuzzer will only cover approximately 24% of the code of a generation-based fuzzer.

6 Conclusions

This paper measures the quantitative differences between mutation and generation-based fuzzing for the PNG im-

age file format and libpng. For this file format, a mutation-based fuzzer is at a large disadvantage due to the lack of diverse files available for testing. Results indicate that large sections of code that will not be exercised. A generation-based approach faired much better. There still exists much future work, including extending this technique to other file formats and decoders and refining the metrics of code coverage.

References

[1] ALEXA. Alexa web search platform: Beta. https://websearch.alexa.com, 2007.

[2] GREEN, A., AND IDEFENSE LABS. notSPIKEfile. http://labs.idefense.com/software/, 2005.

[3] GREEN, A., AND IDEFENSE LABS. SPIKEfile. http://labs.idefense.com/software/, 2005.

[4] RANDERS-PEHRSON, G. Extensions to the PNG 1.1 specification, version 1.1.0. http://www.libpng.org/pub/png/spec, 1998.

[5] ROELOFS, G. libpng home page. http://www.libpng.org/pub/png, 2007.

[6] SUTTON, M., AND IDEFENSE LABS. FileFuzz. http://labs.idefense.com/software/, 2006.

[7] THE GNU PROJECT. gcov – a test coverage program. http://gcc.gnu.org/.

[8] VAN SPRUNDEL, I. Fuzzing: Breaking software in an automated fashion. Talk at: 22nd Chaos Communication Congress: Private Investigations, 2005.

[9] W3C. Portable Network Graphics (PNG) Specification (Second Edition) Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E). http://www.w3.org/TR/PNG, 2003.

[10] WIKIPEDIA. Fuzz testing.