Frontiers in Computing and Intelligent Systems
ISSN: 2832-6024 | Vol. 3, No. 1, 2023

Optimizing WinAFL for Image Parsing Engine Vulnerability Discovery in PDF Readers

Yue Zhang*, Zhibo Du

School of Cybersecurity, Chengdu University of Information Technology, Chengdu, 610200, China
* **Corresponding author**: Yue Zhang (Email: zhangyue@6lack.cn)

Abstract: Fuzzing is a kind of automated vulnerability discovering technique using black-box testing ideas. The PDF file format is very complex and can be embedded in many other formats, providing opportunities for malicious code to hide. In this paper, to solve the problem of high blindness in fuzzing for PDF files by the fuzzing tool WinAFL, we propose a targeted fuzzing scheme for the image parsing engine in PDF readers, optimize WinAFL purposefully, and conduct comparison experiments with the original WinAFL. The experiments show that the optimized fuzzing tool can find an average of 69.43% more unique crashes and 43.28% more path discoveries per unit of time for commonly used PDF readers. So, the method can improve the number of path discoveries and unique crash discoveries, proving the effectiveness and practicality of the method and using this method as an inspiration to propose an improved method for other embedded formats in PDF as the next research direction.

Keywords: Fuzzing; PDF file format; Exploitation of vulnerabilities.

1. Introduction

With the development of anti-virus technology, the spread of malicious executable programs has become very difficult, and hackers prefer to use documents as the first step in implementing attacks in APT attacks. Portable Document Format (a.k.a. PDF) is one of the most popular file formats in the world, and PDF's complex document structure gives malicious code room to hide.

Fuzzing is currently the most popular vulnerability detection technique. Its basic principle is to mutate the seed file randomly to generate a large number of new program inputs, then execute it in the target program, track the target program running state information, and finally analyze the target program crash information to discover program vulnerabilities [1]. Due to the lack of guidance, the blindness of traditional fuzzing seriously affects the efficiency of vulnerability detection. This gave birth to gray-box fuzzing, which obtains program runtime information with the help of lightweight program analysis and guides the fuzzing process with the help of runtime information to improve the efficiency of vulnerability detection [2]

Generally, fuzzing can be divided into generation-based and mutation-based fuzzing. Generative-based fuzzing [3] usually requires the tester to provide the target program input's formatting information or syntax knowledge. The fuzzing program automatically generates program input for vulnerability detection based on the inherent format description files for different target programs. Therefore, this approach is suitable for structured program inputs, such as HTML, JavaScript, etc., and represents fuzzer such as AFLsmart [4]. On the other hand, mutation-based fuzzing does not require familiarity with the format of the input files. Fuzzing tools generate new program input by a set of predefined mutation rules, representing fuzzer such as AFL [5], AFLFast [6], and WinAFL.

In this paper, we optimize the variation method in WinAFL for the parsing engine of image format in PDF parser and propose a targeted fuzzing scheme for the image parsing engine in PDF reader to reduce the blindness of the WinAFL fuzzer for fuzzing of PDF files, which considers the test cases that can explore new execution paths to be of high value.

2. Related Knowledge and Technology

The method of this paper is to realize the vulnerability mining for PDF files on WinAFL, so this section first introduces the general process of the black box path feedback fuzzing test method based on DynamoRIO dynamic binary instrumentation and then briefly introduces the file structure of PDF files.

2.1. WinAFL Based on DynamoRIO

2.1.1. Dynamic binary instrumentation

Dynamic Binary Instrumentation (DBI) [7] is a technique that enables dynamic analysis of binary programs by injecting probe code, which is then executed as regular instructions.

With DynamoRIO, we can monitor the running code of the program, and it also allows us to modify the program's code. To be precise, DynamoRIO is a process virtual machine on which all the code of the monitored program is transferred to the buffer space for simulated execution. The specific architecture is shown in Figure 1.

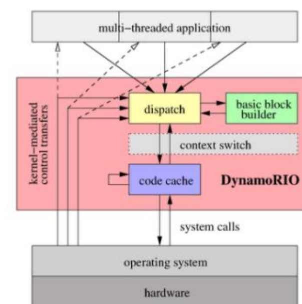

Figure 1. Architecture design of DynamoRIO

Among them, the basic block is an important concept. Suppose all instructions in a monitoring process are split along the boundary of control transfer instructions. In that case, they will be split into many blocks that start with an instruction but end with a control transfer instruction, as

shown in Figure 2.

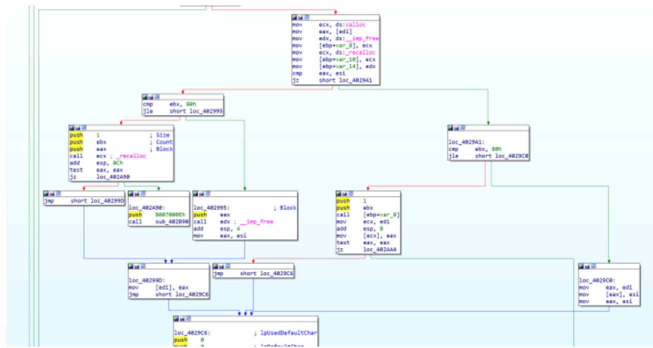

Figure 2. The concept of basic block

These instruction blocks are the basic block concept defined in DynamoRIO, that is, the basic unit of operation. DynamoRIO simulates running instructions in one basic block at a time. When these instructions are finished, they will be run in another basic block through a context switch, and so on, until the monitored process is finished.

2.1.2. AFL Overview

AFL is a coverage-guided grey-box testing tool [5]. Based on genetic algorithm, it uses compile-time instrumentation to determine whether a new internal state of the target program is triggered by the coverage information feedback of the program under test to find interesting test cases and guide the fuzzing strategy, which greatly improves the coverage of the test tool. Its specific approximate steps are shown in Figure 3.

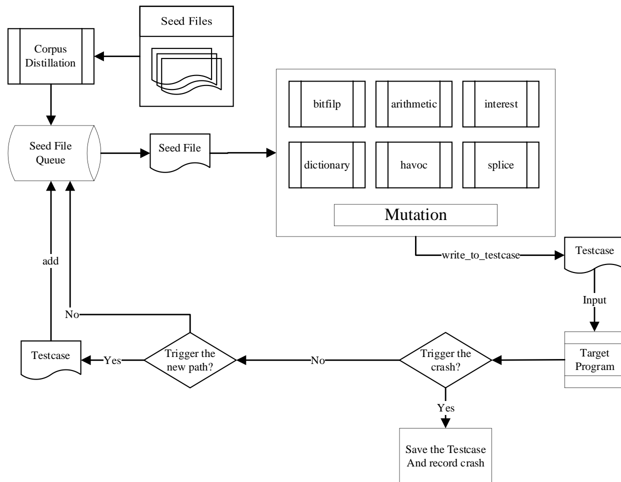

Figure 3. Architecture design of DynamoRIO

1) Use afl-cmin to perform corpus distillation on the original seed files and remove duplicate files based on path feedback. This step is non-essential and is mainly used when there are many initial files. The generated set of files is added to the seed file queue.

2) Select the preferred seed set from the seed queue according to the seed selection algorithm.

3) Select a seed file according to the preset change.

4) For mutating the file using multiple mutation algorithms, many test cases are generated cyclically for testing.

5) If a program crash is triggered, a potential vulnerability may exist, store the file that triggered the crash and record the Crash.

6) If a new path is found, add the test cases for that path to the seed queue.

7) When all the test cases generated by this seed are tested, continue from Step 3 and so on.

2.1.3. WinAFL Fuzzer

In the field of fuzzing, AFL is the most representative fuzzing tool, but because of the reason of its code design, it does not support the Windows platform, and the WinAFL[8] project is the transplant of this Fuzzer on the Windows platform. AFL uses compile-time instrumentation and genetic algorithm to implement its functions, while WinAFL uses DynamoRIO dynamic instrumentation instead of AFL compile-time instrumentation to adapt to the Windows platform, which is mainly closed-source software.

Through binary dynamic instrumentation, DynamoRIO can feed back the coverage information of the binary program to WinAFL, and the coverage is passed through the pipeline. In general, the whole WinAFL execution process is roughly as follows:

1) afl_fuzz.exe interacts with the target process by creating a named pipe and memory mapping. The pipe is used to send and receive commands to interact with the other process, and the memory mapping is mainly used to record the coverage information;

2) Coverage record is mainly through drmgr_register_bb_instrumentation_event to set the callback function executed by BB. The coverage is recorded by instrument_bb_coverrage or instrument_edge_coverage, and if new execution paths are found, the samples are placed into the queue directory for subsequent file mutation to improve code coverage.

3) After the target process executes to the target function, pre_fuzz_handler will be called to store context information, including registers and running parameters;

4) After the target function is launched, the post_fuzz_handler function will be called to record the reply context information so as to execute the original target function and go back to the second step;

5) When the number of target function runs reaches the number of specified loop calls, the process will be interrupted and exit.

2.2. PDF

2.2.1. Physical Structure

PDF documents are constructed hierarchically from a set of interconnected objects, according to the Adobe PDF Reference [9]. Its physical structure consists of four basic components, as shown in Figure 4.

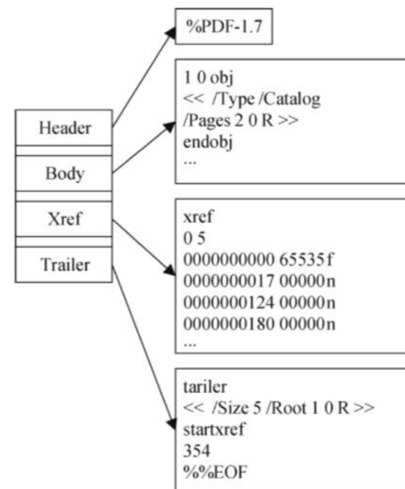

Figure 4. The physical structure of a PDF document

1) Header: The type identification of a PDF document is

located in the header, which starts with "%PDF-" followed by the version of the PDF specification that the document conforms to. For example, "%PDF-1.7" indicates that the document conforms to the PDF 1.7 version specification.

2) Body: The main content of the document is stored in the body, which consists of multiple indirect objects that can store text, images, and other content. Indirect objects can also record information such as font, color, size, image content, layout, position, and the relationship and display order between objects. When storing this content, PDF documents generally use existing compression methods to save storage space and document transmission costs.

3) Cross-reference table (Xref): Starting with the keyword "Xref", this section records the byte offsets of each object relative to the starting position of the document, in bytes.

4) Trailer: Starting with "trailer" and ending with "%%EOF", the trailer stores two important pieces of information: the root object identified by the attribute keyword "Root", and the byte offset position of the cross-reference table relative to the beginning of the document, identified by the keyword "startxref".

2.2.2. Logical Structure

PDF documents are a tree-like logical structure, with the root node "Catalog" serving as the connection point between the physical and logical structure of the document, as shown in Figure 5.

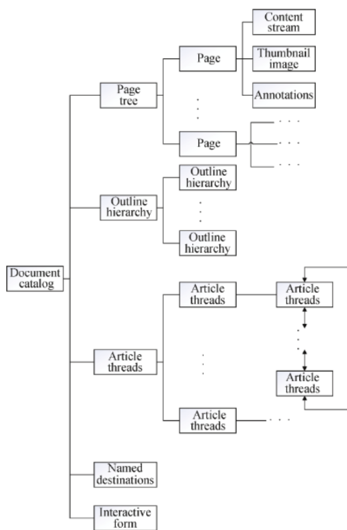

Figure 5: The logical structure of a PDF document

When a PDF reader opens a PDF document, it first locates the Catalog from the Root field in the Trailer and then uses this node to parse the page, directory, and link information. The Catalog contains a wealth of information, including a Page tree, an Outline hierarchy, and Article threads. The Page tree is a collection of descriptions for all the pages in the PDF document, used to organize all Page objects. The Page object, in turn, describes the attributes, resources, and other information specific to a PDF page.

3. Optimizing Fuzzing for PDF File

This article proposes an optimization method for WinAFL, which involves hijacking the "write_to_testcase" function in the main fuzzer part of the WinAFL project to achieve mutation of the image section in PDF documents. By using this method, more efficient and accurate testing can be carried out for the image parsing part in PDF parsing software, thereby improving the efficiency and quality of fuzzing test cases.

3.1. The write_to_testcase Function

The write_to_testcase function is a key function in WinAFL. It is used to write the test cases generated by the fuzzer to disk for subsequent execution and analysis.

The function is located in the fuzzer.c file of WinAFL, and its prototype is as follows:

void write_to_testcase(void* mem, size_t len, char* filename)

In the process of fuzzing with WinAFL, the main fuzzer will call the write_to_testcase function to save the test case to the disk for each execution, so as to facilitate subsequent analysis and reproduction.

In WinAFL, the write_to_testcase function is often executed after file mutation. Its main function is to read the binary stream of the seed file that is ready to be executed for testing into memory and write this part of the binary stream into the current input file, which is by default the ".cur_input" file specified by WinAFL's output directory, but can also be a specified file. Then WinAFL will call the run_target function to launch the target program and input this file into the target program.

4. Algorithm Framework

In general, when performing fuzzing on PDF files, the common method is to use PDF files as seed files. In this article, in order to mutate the image part, the seed file is replaced with an image format. Then, the write_to_testcase function is hijacked to replace the original operation of directly inputting the binary stream into the ".cur_input" file with custom operations. Specifically, after hijacking the write_to_testcase function, a custom script is used to manipulate the test case so that each test case written to disk contains a mutated image. Then, the run_target function was called to input the embedded PDF file into the target program. The modified process is shown in Figure 6.

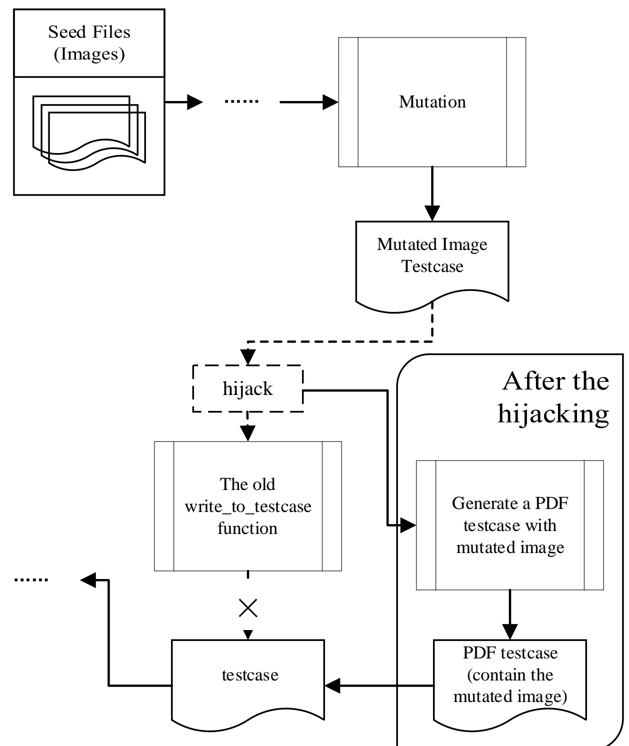

Figure 6: Schematic diagram of the algorithm flow

Through the function hijacking operation described above, the range of file mutation in the PDF file can be successfully narrowed down to a specific part, thereby improving the quality of test cases.

To implement the above method and achieve better scalability, I chose to use a mixed compilation of Python and C language. Python has a large number of powerful library functions, and the code is concise and easy to maintain. At the same time, it can be easily embedded in C language.

5. Experiment and Analysis

This paper improves fuzzing method based on WinAFL 1.16b (Based on AFL 2.43b). The experiment was deployed on a VMware virtual machine with 8GB memory, 8 cores, and 12th Gen Intel(Rz) Core(TM) i7-12700H CPU running 64-bit Windows 21H2. WPS PDF reader and pdf-x-change were selected as the targets, with each program running for 24 hours and the test results taken as the average of 3 tests. The experimental results for these two target programs are shown in Table 1.

Table 1. Comparison of experimental results

Target Program	Fuzzer	Total Paths	Comparison	Unique Crashes	Comparison
WPS PDF reader	WinAFL	513	+40.94%	35	+74.29%
	Optimized WinAFL	723		61	
pdf-x-change	WinAFL	833	+45.62%	175	+64.57%
	Optimized WinAFL	1213		288	

The results show that the optimized WinAFL outperforms the native WinAFL in both the number of crashes discovered and the number of paths discovered for these two target programs. For WPS PDF reader, the number of paths discovered increased by 40.94%, and the number of crashes discovered increased by 26. For pdf-x-change, the number of paths discovered increased by 45.62%, and the number of crashes discovered increased by 113. Overall, the effectiveness of this optimization method is significant, which demonstrates the feasibility of the approach presented in this paper.

6. Conclusion

The purpose of this paper is to improve the fuzzing method in order to increase its efficiency and accuracy for testing PDF files. In order to mutate the image part of PDF files, this paper adopts the method of replacing the seed file with image format and controlling the test case generation process by hijacking the write_to_testcase function. In the implementation, the optimization method was implemented by using a mixed compilation of Python and C, while WPS PDF reader and pdf-x-change were later selected as the target programs for testing. The experimental results show that the method proposed in this paper has a significant improvement effect, proving its feasibility and practicality in the field of PDF file testing. For the subsequent research direction, I will use similar ideas to achieve the improvement of fuzzing methods for PDF documents in "otf", "ttf" and other font file formats, and will also try to use the overall change of PDF structure. In addition, I will try to use the overall variation of PDF structure to enrich the diversity of PDF document variants.

References

[1] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” Cybersecurity, vol. 1, no. 1, pp. 1–13, 2018.

[2] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, “A systematic review of fuzzing techniques,” Computers & Security, vol. 75, pp. 118–137, 2018.

[3] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in C compilers,” in Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, 2011, pp. 283–294.

[4] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, “ Smart greybox fuzzing ” IEEE Transactions on Software Engineering, vol. 47, no. 9, pp. 1980–1997, 2019.

[5] M. Zalewski, "American fuzzy lop (afl) - a security-oriented fuzzer," Online. Available: http://lcamtuf.coredump.cx/afl/. [Accessed: Mar. 6, 2023].

[6] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 1032–1043.

[7] DynamoRIO Dynamic Instrumentation Tool Platform. Available online: https://dynamorio.org/page_home.html (accessed on March 6, 2023)

[8] “WinAFL.” Google Project Zero, Mar. 06, 2023. Accessed: Mar. 07, 2023. [Online]. Available:

https://github.com/googleprojectzero/winafl

[9] Adobe Systems Incorporated. (2018). PDF Reference (sixth edition): Adobe Portable Document Format version 1.7 [Online]. Available:

https://www.adobe.com/devnet/pdf/pdf_reference_archive.html.