IT in Industry, vol. 2, no. 3, 2014 Published online 27-Oct-2014

A Smart Fuzzing Approach for Integer Overflow Detection

Jun Cai, Peng Zou, Jun He
The Academy of Equipment
Beijing, China
cjgfkd@163.com

Jinxin Ma
China Informaiton Technology Security Evaluation Center
Beijing, China
majinxin2003@126.com

***Abstract*—Fuzzing is one of the most commonly used methods to detect software vulnerabilities, a major cause of information security incidents. Although it has advantages of simple design and low error report, its efficiency is usually poor. In this paper we present a smart fuzzing approach for integer overflow detection and a tool, SwordFuzzer, which implements this approach. Unlike standard fuzzing techniques, which randomly change parts of the input file with no information about the underlying syntactic structure of the file, SwordFuzzer uses online dynamic taint analysis to identify which bytes in the input file are used in security sensitive operations and then focuses on mutating such bytes. Thus, the generated inputs are more likely to trigger potential vulnerabilities. We evaluated SwordFuzzer with an example program and a number of real-world applications. The experimental results show that SwordFuzzer can accurately locate the key bytes of the input file and dramatically improve the effectiveness of fuzzing in detecting real-world vulnerabilities.**

Keywords*—information security; vulnerability detection; dynamic taint analysis; smart fuzzing

I. INTRODUCTION

Software security has become a very import part of information security in recent years. Vulnerabilities are one of the root causes of security problems. Once they are exploited by attackers, they may cause serious damages. Therefore, vulnerability detection technology is gaining more and more attention in the field of information security.

Vulnerability is a hole or a weakness in the application, which can be a design flaw or an implementation bug. Vulnerability allows an attacker to cause harm to the stakeholders of an application. Stakeholders include the application owner, application users, and other entities that rely on the application [1]. There are many kinds of software vulnerabilities, each has its special cause. It is impossible to find a detection approach which can detect all kinds of vulnerabilities. In this paper, we focus on how to detect integer overflow vulnerability.

As shown in Table I, the annual number of identified integer overflow vulnerabilities recorded by the National Vulnerability Database (NVD [2]) is stable at around 110 since 2007. Moreover, the proportion of integer overflow vulnerabilities in the total annual number of all vulnerabilities is also stable at around 2%. Although this proportion does not seem high, these vulnerabilities often have the highest severity (with score 7~10). How to detect and eliminate integer overflow vulnerability has becoming a hot research topic.

There are several techniques and tools to detect integer overflows, such as KLEE [3], SAGE [4], BuzzFuzz [5] and so on. KLEE performs symbolic execution to detect integer overflows, while SAGE and BuzzFuzz perform white box fuzzing. Though they are all famous and effective tools, they all rely on source code which is not always available to users. IntScope [6] is an excellent tool which performs binary-based fuzzing, but it relies on static analysis. Our aim was to develop a dynamic binary-based fuzzing method that is efficient for integer overflow vulnerability detection.

In this paper, we present a smart fuzzing approach for integer overflow detection and implement a tool called SwordFuzzer which implements this approach. SwordFuzzer currently works with x86 binaries on Linux and targets file processors, while we plan to extend it to run on the Windows operating system and target network packet processors.

The key ideas of our approach are: (1) using taint analysis to identify the key bytes of input files that affect the security sensitive operations of the target application; (2) focusing on mutating those key bytes to trigger potential vulnerabilities.

Our contributions can be summarized as follows:

(1) We propose an effective dynamic integer overflow vulnerability detection method combining fuzzing with taint analysis.

(2) We implement a prototype called SwordFuzzer which can perform fast online taint analysis and automatic fuzzing for real-world binaries.

TABLE I. THE REPORTED ANNUAL NUMBER OF INTEGER OVERFLOWS AND ALL VULNERABILITIES IN NVD FROM 2007 TO 2013

Year	Number of Integer Overflows	Number of All Vulnerabilities	Proportion
2013	96	5186	1.85
2012	104	5289	1.97
2011	92	4150	2.22
2010	112	4639	2.41
2009	129	5732	2.25
2008	112	5632	1.99
2007	126	6514	1.93

The remainder of this paper is organized as follows: Section 2 introduces the character of integer overflow, the basic ideas of taint analysis and smart fuzzing. Section 3 presents the design and implementation of SwordFuzzer. Section 4 evaluates SwordFuzzer. Section 5 examines the limitations of the current implementation, along with future considerations. Then related work is presented in Section 6 and finally conclusion in Section 7.

II. Overview

A. *Integer Overflow*

Integer overflow errors occur when a program fails to account for the fact that an arithmetic operation can result in a quantity either greater than a data type's maximum value or less than its minimum value. These errors often cause problems in memory allocation functions, where user input intersects with an implicit conversion between signed and unsigned values. If an attacker can cause the program to under-allocate memory or interpret a signed value as an unsigned value in a memory operation, the program may be vulnerable to a buffer overflow [7].

An integer overflow vulnerability usually has the following features: (1) Untrusted source. All the data comes from users can be treated as taint source, such as files, network packages, keyboard input and so on. (2) Various types of sinks. Sinks are the security sensitive points of the target application. If an overflowed value is used in these points, a vulnerability may occur. For example, when an integer is used in the determination of an offset or size for memory allocation, copying, concatenation, or similarly, if the integer is incremented past the maximum possible value, it may wrap to become a very small, or negative number, therefore providing an incorrect value. (3) Incomplete or improper sanitization checks. Almost all the subtle integer overflow vulnerabilities are actually caused by incomplete or improper checks [6, 7].

As a motivating example, Fig. 1 shows the source code of a simple file processor which is vulnerable. It just reads an integer (four bytes) from a file “123.data”. As Fig. 1 shows, the value of the integer variable “a” depends on the 6~9th bytes of the input file (line 9, 10) while “size” depends on “a” (line 12). Because “size” is an unsigned short variable, it may be overflowed by “a”, then allocated memory size of “buf” (line 15) will not be enough for “memcpy” (line 16), thus an integer overflow occurs.

B. *Taint Analysis*

Taint analysis [8-12] is an emerging program analysis technique which has been widely used in many fields of information security in recent years, such as malicious code analysis, network attack detection and protection, software vulnerability detection, protocol format reverse analysis, and so on. The main idea behind taint analysis is that any variable that can be modified (directly or indirectly) by the user can become a security vulnerability (the variable becomes tainted) when a tainted variable is used to execute dangerous commands.

We can distinguish two taint analysis approaches: static taint analysis and dynamic taint analysis. The former is performed mostly at source level by means of abstract interpretation techniques. However, most of the time, the source codes of applications are not available. Meanwhile, static taint analysis may generate a lot of false positives. So, dynamic taint analysis is the more commonly used approach, but it is very complex to implement.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[]){
4     unsigned short size;
5     int a;
6     char *buf;
7     FILE *fp;
8     fp = fopen(argv[1], "rb");
9     fseek(fp,5L,0);
10    fread(&a, 1, 4, fp);
11    fclose(fp);
12    size = a;
13    printf("%x\n",a);
14    printf("%x\n",size);
15    buf= (char *)malloc(size*sizeof(char));
16    memcpy(buf, "hello", a);
17    return 0;
18 }
```

Fig. 1. Source code of an example file processor: integer.c

Dynamic taint analysis usually involves three steps: taint introduction, taint propagation, and taint check. The advantages of dynamic taint analysis are that it offers the capabilities to detect most of the input validation vulnerabilities with a very low false positive rate. While the disadvantages are that it is generally suffering from slow execution, and the problems are detected only for the executions path that have been executed until now (not for all executable paths) which can lead to false negatives. This paper focuses on solving the slow execution problem and the correctness of taint propagation.

C. *Smart Fuzzing*

Fuzzing [13, 14] is a traditional vulnerability detection technique. It was first proposed and used by Barton Miller in 1989. Although it was invented more than 20 years ago, it does not obsolete and is still an important and commonly used method. The idea behind fuzzing is very simple: generating malformed inputs and feeding them to an application; if the application crashes or hangs, a potential vulnerability is detected. Fuzzing has been discussed extensively in both academia and industry and has proven successful in finding vulnerabilities [15].

We use smart fuzzing to distinguish from standard fuzzing. The prefix smart implies that fuzzing is not performed purely randomly, but by taking advantage of some priori knowledge, which can be the input formats, some results obtained from preliminary analysis of the software, or even some information gained during the fuzzing process [16-18]. The goal is then to generate (or mutate) inputs that are more likely to trigger potential vulnerabilities.

With the expansion of the scale of software and developers’ increasing emphasis on software security, it is becoming more and more difficult to detect modern software’s vulnerabilities by means of simple purely fuzzing. Traditional fuzzing is facing the following challenges: (1) Blindly selection of mutation location and simple mutation policy; (2) Poor code coverage; (3) Obstacle brought by the internal check mechanisms of software. This paper focuses on solving the first challenge.

III. Design and implemention

In this section, we present the design and implementation of SwordFuzzer. We first give an overview of SwordFuzzer and then describe its detailed design.

The intuition behind our approach is that inter-overflow vulnerabilities are usually caused by the misuse of overflowed values. For example, the overflowed value is used in memory allocation functions (e.g., malloc, calloc, realloc) as a size argument, and it usually results in an insufficient memory allocation, which may eventually become an attacker's springboard to a buffer overflow. Thus, we check all the dangerous functions to see whether their parameters are tainted. If there is any dangerous function like malloc, which uses tainted data as parameters, a potential vulnerability may exist. Then we use guided fuzzing to find out where the suspicious vulnerability lies.

At a higher level, SwordFuzzer takes an executable application and a seed file as input, if the input application has an integer overflow vulnerability, it will output the vulnerability's POC (Proof of Concept) and some other information which may be very valuable for us to illustrate the vulnerability. Fig. 2 shows the overview of SwordFuzzer. Given a binary program P to be analyzed, SwordFuzzer's working process can be divided into two procedures:

Taint Analysis Procedure The taint analysis procedure contains three modules: Binary Instrumentation Tool, Taint Tracer and KeyBytes Identifier. The Binary Instrumentation Tool provides us with a basis platform for binary analysis; The Taint Tracer implements our online fine-grained dynamic taint tracing logic, it can output a report on whether there are tainted date being used in some dangerous ways, a call graph of the taint propagation process and all the taint propagation instructions. The KeyBytes Identifier is an automatic shell script which can further identify which bytes in the input file are used by the sensitive functions.

Fuzzing Procedure The taint analysis procedure reports us that there may be a potential integer overflow vulnerability in the target application. For example, it reports that some bytes of the input file affect the parameter of a malloc function. Then, in the fuzzing procedure, we first use the KeyBytes Mutater to mutate those bytes to generate new files which we

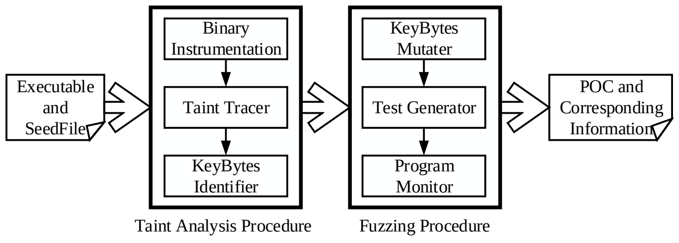


Fig. 2. An overview of SwordFuzzer

call test cases, we then use the Test Driver to feed these test cases to the target program, meanwhile, we use the Program monitor to keep watch on the target program's behaviors, if the program crashes or hangs, there may be a integer overflow vulnerability, and the corresponding test case is just the vulnerability's POC.

A. *Binary Instrumentation*

We implement our taint analysis logic by the aid of Pin DBI (dynamic binary instrumentation) framework. Pin [24] is a DBI framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools. The tools created using Pin, called Pintools, can be used to perform program analysis on user space applications in Linux and Windows.

Briefly, Pin consists of a virtual machine (VM) library, and an injector that attaches the VM in already running processes or new processes that launch by themselves. Pintools are shared libraries that employ Pin's extensive API to inspect and modify a binary at the instruction level.

B. *Taint Tracer*

Taint Tracer is the core module of the taint analysis procedure. Its function is to perform dynamic taint tracing. The detailed design is as flows:

1) *Taint Introduction*

There are usually two ways to introduce taint. The first way is to hook a system function or system call, for instance, if we define files as taint source, the fread() call in Fig. 1 would introduce taint. The other way is to introduce taint during the process of program execution which can be customized by users, for example, when the program executes to a certain function customized by the user, the corresponding taint would be introduced.

Our Taint Tracer introduces taint by hooking system calls. For example, if files are defined as taint sources, it will hook open, create, read and so on; if network packages are defined as taint sources, it will hook socket calls.

Meanwhile, our Taint Tracer allows user to customize the taint sources by a series of parameters. For file taint source, it has three parameters: "-tf", "-to" and "-ts". The "-tf" parameter indicates the input file to be set as taint source file, the "-to" parameter indicates the offset in the taint file to be set as taint, and the "-ts" parameter indicates how much bytes to be set as taint source.

2) *Taint Propagation*

During program execution, tainted dates are tracked as they are copied and altered by program instructions. Tainted data are assigned with tags. Tags propagate from the instructions' source operand to destination operand according to the semantic of the instruction. The taint propagation algorithm directly affects the correctness of the whole taint propagation process.

There are two basic problems of taint propagation. One problem is the taint propagation granularity, which refers to the size of the smallest taint unit. It is important to choose a

```
VOID CallocCheck(ADDRINT n, ADDRINT size)
{
      if(IsTainted(n))
            TraceFile <<"Calloc: n= "<<*(int *)n<< " is tainted!" << endl;
      if(IsTainted(size))
            TraceFile<<"Calloc: size= "<<*(int *)size<< " is tainted!"<< endl;
}
```

Fig. 3. Example Pin analysis call

suitable granularity. If the granularity is too large, it would easily lead to tainted data loss and inaccurate taint propagation. On the contrary, if the granularity is too small, it would easily lead to excessive taint proliferation and the taint propagation logic would become too complicated to implement. Therefore, we choose byte-level taint propagation granularity, since most instructions are calculated in bytes and the smallest addressable chunk of memory in most architectures is a byte. Our choice is sufficient for fine-grained taint tracing.

Another problem is the taint tag size. Every unit of tainted data is assigned with a taint tag. The smallest size of taint tag can be just a single-bit. For example, “1” indicates that the unit of data is tainted while “0” indicates untainted. Larger tags are more versatile as they allow for different types of data to be tagged uniquely. However, larger tags require more complicated propagation logic and more storage space. Since our goal is fast taint tracing and the faster the better, we choose sing-bit tags.

At binary level, two objects can be tainted: memory locations and registers, therefore, our Taint Tracer stores taint tags in two data structures. One data structure is the “mem_taint_map”, it holds one bit for each byte of process addressable memory. Another data structure is the “reg_taint_map”, it assigns one 32-bit unsigned integer for each 32-bit register of the x86 architecture, the lower 4 bits of the integer represent the register’s tags.

As for the policy of taint propagation, we mainly consider three types of propagation: (1) Data replication operations (e.g. MOV, PUSH, POP, etc.). For this case, just copy the tags of the source operand to the destination operand; (2) Data arithmetic operations and bit manipulation (e.g. ADD, SUB, AND, SHL, etc.). For this case, we need to do arithmetic operations on the tags of both the source operand and the destination operand, and update the destination operand’s tag with the operational result. (3) Side effect. Some operations may cause hidden side effects, for instance, the REP operation would affect the ECX register and arithmetic operations would affect EFLAGS register.

3) *Taint Check*

The purpose of taint check is to check whether tainted data are used in some dangerous ways, such as covering the return address on the stack, being the value of EIP, being the parameter of dangerous functions (e.g. malloc).

Currently, we mainly check the two types of sensitive operation: memory allocation and branch statement. For the first type, we check the taint tags of the addresses of the parameters of malloc, calloc and realloc to see if they are tainted; for the second type, we instrument the jmp/call/ret instructions to see if the address of the branch target is tainted. Fig. 3 shows an example Pin analysis call to check “calloc”.

C. *Keybytes Identifier*

The key idea of the Keybytes Identifier is to trace every single byte of the input taint file to find out all the bytes directly affect the sensitive operations which we call “Keybytes”. We use a shell script to implement this idea. We run the taint tracer with the “-to” parameter equals to “0” and the “-ts” parameter equals to “1” at the first time, then we run the taint tracer again and again with the “-to” parameter equals to “1”, and so on.

D. *Keybytes Mutater*

The Keybytes Mutater’s function is to generate test cases by the means of mutating the key bytes of the input seed file. We use a series of mutation policies to mutate these bytes. For example, we may mutate one bytes a time or a several bytes a time, and we may want to get different mutations for every single byte. All the policies can be configured by users via several parameters.

E. *Test Driver*

The Test Driver’s function is to feed the test cases to target program one by one, and notice the Program Monitor to monitor the behavior of target program execution.

F. *Program Monitor*

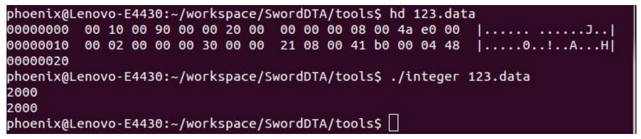
The Program Monitor’s function is to monitor and capture program exceptions. Monitoring target under test is a critical step because it helps identifying and analyzing which test cases cause program exceptions and why the exceptions happen. Our Program Monitor is a debugger which can attach to the program under test.

IV. EVALUATION

In this section, we first evaluated SwordFuzzer with the example application showed in Fig. 1. Then we evaluated it with a real-world vulnerable application. Finally we evaluated it for testing a variety of real-world applications. All the tests were done on Ubuntu 12.04. We mainly evaluated its ability to accurately identify key bytes, to generate effective test cases via mutating the key bytes and to detect real-world vulnerabilities.

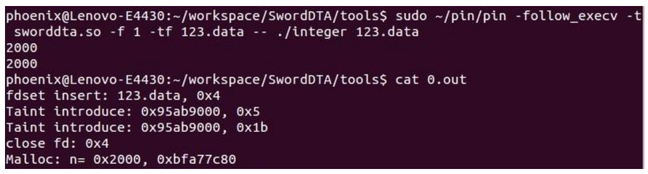
A. *Vulnerability Dection of Example Application*

We used “gcc” to compile the source code showed in Fig. 1 to an object file “integer” for test. Firstly, we gived it a seed input file “123.data” and it run normally (see Fig. 4). Then we used SwordFuzzer testing this program. The result is that SwordFuzzer identified the “6~9” bytes of the file “123.data” affecting the sensitive operation “malloc” (see Fig. 5), this result is obviously accurate. Then, in the next fuzzing procedure, we kept on mutating the key bytes of the seed file to generate new test cases and successfully found a test case triggering the integer overflow vulnerability soon(see Fig. 6).



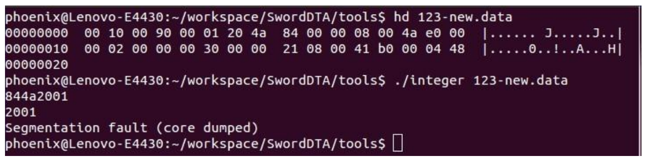
```
phoenix@Lenovo-E4430:~/workspace/SwordDTA/tools$ hd 123.data
00000000  00 10 00 90 00 00 20 00  00 00 00 08 00 4a e0 00  |...... ......J..|
00000010  00 02 00 00 00 30 00 00  21 08 00 41 b0 00 04 48  |.....0..!..A...H|
00000020
phoenix@Lenovo-E4430:~/workspace/SwordDTA/tools$ ./integer 123.data
2000
2000
phoenix@Lenovo-E4430:~/workspace/SwordDTA/tools$ 
```

Fig. 4. The execution of "integer" with an seed file "123.data"



```
phoenix@Lenovo-E4430:~/workspace/SwordDTA/tools$ sudo ~/pin/pin -follow_execv -t
 sworddta.so -f 1 -tf 123.data -- ./integer 123.data
2000
2000
phoenix@Lenovo-E4430:~/workspace/SwordDTA/tools$ cat 0.out
fdset insert: 123.data, 0x4
Taint introduce: 0x95ab9000, 0x5
Taint introduce: 0x95ab9000, 0x1b
close fd: 0x4
Malloc: n= 0x2000, 0xbfa77c80
```

Fig. 5. A taint analysis result of "integer"



```
phoenix@Lenovo-E4430:~/workspace/SwordDTA/tools$ hd 123-new.data
00000000  00 10 00 90 00 01 20 4a  84 00 00 08 00 4a e0 00  |...... J.....J..|
00000010  00 02 00 00 00 30 00 00  21 08 00 41 b0 00 04 48  |.....0..!..A...H|
00000020
phoenix@Lenovo-E4430:~/workspace/SwordDTA/tools$ ./integer 123-new.data
844a2001
2001
Segmentation fault (core dumped)
phoenix@Lenovo-E4430:~/workspace/SwordDTA/tools$ 
```

Fig. 6. A test case triggering the vulnerability

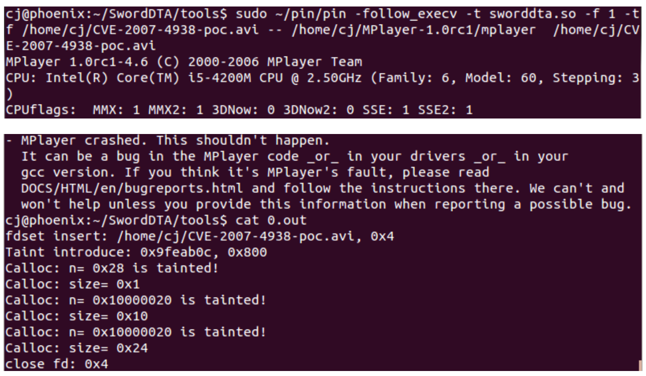
B. *Vulnerbility Dection of Real-world Application*

In this section, we used a known real-word vulnerability to evaluate SwordFuzzer, that is CVE-2007-4938 [25], CVE-2007-4938 is an integer overflow vulnerability of MPlayer 1.0rc1. The POC was get from the Exploit Databases [26].

As Fig. 7 shows, SwordFuzzer found the parameter "n" of three calloc() functions is tainted data. For the last calloc() function, the value of "n" is "0x10000020" while the value of the parameter "size" is "0x24", thus the size of the memory to be allocated would be 0x10000020*0x24=0x240000480; this value exceeds the maximum possible value of an integer, thus an integer overflow occurs; Furthermore, SwordFuzzer identified that it is the "225~228" bytes of the input POC file that affected this calloc().

C. *Testing Results*

Next, we evaluated SwordFuzzer with a number of widely used utility applications. Table 2 shows the taint analysis result which indicates that our tool can give alarms of integer overflows and can accurately location the key bytes of various types of files such as png, gif, jpg, bmp and so on. We also successfully generate test cases for these applications. Take the picture viewer application "eog" as an example, we choose a common gif file with a size of 193885 bytes (193.9KB) as seed file, and we find out that there are only five key bytes (16, 27~30) affecting sensitive operations, that is, we need only mutate 5 bytes instead of 193885 bytes, thus the efficiency of fuzzing has been improved dramatically.



```
cj@phoenix:~/SwordDTA/tools$ sudo ~/pin/pin -follow_execv -t sworddta.so -f 1 -t
f /home/cj/CVE-2007-4938-poc.avi -- /home/cj/MPlayer-1.0rc1/mplayer  /home/cj/CV
E-2007-4938-poc.avi
MPlayer 1.0rc1-4.6 (C) 2000-2006 MPlayer Team
CPU: Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz (Family: 6, Model: 60, Stepping: 3
)
CPUflags:  MMX: 1 MMX2: 1 3DNow: 0 3DNow2: 0 SSE: 1 SSE2: 1
```

```
- MPlayer crashed. This shouldn't happen.
  It can be a bug in the MPlayer code _or_ in your drivers _or_ in your
  gcc version. If you think it's MPlayer's fault, please read
  DOCS/HTML/en/bugreports.html and follow the instructions there. We can't and
  won't help unless you provide this information when reporting a possible bug.
cj@phoenix:~/SwordDTA/tools$ cat 0.out
fdset insert: /home/cj/CVE-2007-4938-poc.avi, 0x4
Taint introduce: 0x9feab0c, 0x800
Calloc: n= 0x28 is tainted!
Calloc: size= 0x1
Calloc: n= 0x10000020 is tainted!
Calloc: size= 0x10
Calloc: n= 0x10000020 is tainted!
Calloc: size= 0x24
close fd: 0x4
```

Fig. 7. Test result of a known real-world vulnerability

TABLE II. THE REPORTED ANNUAL NUMBER OF INTEGER OVERFLOWS AND ALL VULNERABILITIES IN NVD FROM 2007 TO 2013

Program	**File Type**	**File Size**	**Key Bytes**	**Affected sensitive operations**
eog 3.4.2	png	105274B	17~25, 50~53	malloc
eog 3.4.2	gif	193885B	16, 27~30	malloc, realloc
lowriter 3.5.7.2	odt	23366B	Null	Null
lowriter 3.5.7.2	docx	106220B	Null	Null
FoxReader 1.1	pdf	228667B	Null	Null
Shotwell 0.12.3	jpg	105066B	23~24	malloc,calloc, realloc
Shotwell 0.12.3	bmp	304902B	19~26	malloc,calloc, realloc

V. LIMATATINOS AND FUTURE WORK

The limitations of our work can be summarized as follows: (1) Incomplete taint tracing. SwordFuzzer is implemented based on Pin DBI framework, which could only perform program analysis on user space, thus may lose some taint information in the kernel space. (2) Not support 64-bit architectures.

In the future, we plan to extend SwordFuzzer to run on Windows operating systems and 64-bit architectures, and target network packet processors.

VI. RELATED WORK

Zzuf [19] is a transparent application input fuzzer. It works by intercepting file and network operations and changing random bits in the program's input. Zzuf's main purpose is to make things easier and automated.

Sulley [20] is an actively developed fuzzing engine and fuzz testing framework consisting of multiple extensible components. The goal of the frame work is to simplify not only data representation but to simplify data transmission and instrumentation.

Peach [21] is a Smart Fuzzer that is capable of performing both generation and mutation based fuzzing. One distinctive high level concept of peach is modeling. Peach operates by applying fuzzing to models of data and state.

Fuzzgrind [22] is a fully automatic fuzzing tool, generating test files with the purpose of discovering new execution paths likely to trigger bugs and potentially vulnerabilities. It is based on the concept of symbolic execution.

TaintScope [23] is an automatic fuzzing system working at the x86 binary level using dynamic taint analysis and symbolic execution techniques. The most amazing feature of TaintScope is that it can generate checksum-aware inputs.

VII. CONCLUSION

In this paper, we have presented a taint based smart fuzzing approach for integer overflow vulnerability detection. Our approach first uses dynamic taint analysis to find suspicious integer overflow vulnerabilities, then uses fuzzing to validate them. We have implemented our approach in a system called SwordFuzzer. Experimental results show that SwordFuzzer can accurately locate the key bytes of the input file and generate effective fuzzing test cases via mutating these key bytes. For the number of key bytes occupies only a small proportion of file size, thus the efficiency of fuzzing has been improved dramatically.

ACKNOWLEDGMENT

This research has been supported by National High Technology Research and Development Program of China (2012AA012902) and "HGJ" National Major Technological Projects (2013ZX01045-004-002).

REFERENCES

[1] OWASP, *Category:Vulnerability* [Online]. Available: https://www.owasp.org/index.php/Category:Vulnerability

[2] NIST, National *Vulnerability Database* [Online]. Available: http://web.nvd.nist.gov/view/vuln/search-advanced

[3] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, 2008, pp. 209-224.

[4] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008, pp. 151-166.

[5] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing", in *Proceedings of the IEEE 31st International Conference on Software Enineering* (*ICSE'09)*, May 16-24, 2009, Vancouver, Canada, pp. 474–484.

[6] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: automatically detecting integer overflow vulnerability in X86 binary using symbolic execution", in *Proceedings of the 16th Network and Distributed System Security Symposium (NDSS'09)*, San Diego, CA, February 2009.

[7] OWASP, *Integer Overflow* [Online]. Available: https://www.owasp.org/index.php/Integer_overflow

[8] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005).*

[9] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework", in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*, ACM, July 9–12, 2007, London, England, United Kingdom, pp. 196-206.

[10] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)", in the *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 317-331.

[11] E. Bosman, A. Slowinska, and H. Bos, "Minemu: the world's fastest taint tracker," in *Proceedings of the 14th International Conference on Recent advances in Intrusion Detection (RAID'11)*, 2011, pp. 1–20.

[12] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *VEE'12*, March 3–4, 2012, London, England, UK.

[13] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Addison–Wesley Professional, United States, 2007.

[14] A. Takanen. (2009). *Fuzzing: the past, the present and the future*, [Online]. Available: http://actes.sstic.org/SSTIC09/Fuzzing-the_Past-the_Present_and_the_Future/SSTIC09-article-A-Takanen-Fuzzing-the_Past-the_Present_and_the_Future.pdf

[15] B. S. Pak, "Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution", School of Computer Science Carnegie Mellon University, May 2012.

[16] S. Rawat and L. Mounier, "Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: few preliminary results", in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 531-533.

[17] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation, (ICST)*, 2011, pp. 427–430.

[18] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A taint based approach for smart fuzzing," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 818–825.

[19] Caca labs, *Zzuf – Multi-purpose fuzzer* [Online]. Available:. http://caca.zoy.org/wiki/zzuf

[20] *A pure-python fully automated and unattended fuzzing framework* [Online]. Available: https://github.com/OpenRCE/sulley

[21] M. Eddington, *Peach fuzzer* [Online]. Available: http://peachfuzzer.com/

[22] Sogeti ESEC Lab, *Fuzzgrind* [Online]. Available: http://esec-lab.sogeti.com/pages/Fuzzgrind

[23] T. Wang, T. Wei, G. Gu, W. Zou, "Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution", *ACM Transactions on Information and System Security*, vol. 14, no.2, article 15, September 2011.

[24] Intel, *Pin - A Dynamic Binary Instrumentation Tool* [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[25] NIST, *CVE-2007-4938* [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-4938

[26] *Offensive Security, The Exploit Database* [Online]. Available: http://www.exploit-db.com/