Research Article

An Exploitability Analysis Technique for Binary Vulnerability Based on Automatic Exception Suppression

Zhiyuan Jiang, Chao Feng , and Chaojing Tang

School of Electronic Science, National University of Defense Technology (NUDT), Changsha, Hunan, China

Correspondence should be addressed to Chao Feng; chaofeng@nudt.edu.cn

Received 12 October 2017; Revised 9 January 2018; Accepted 11 February 2018; Published 28 March 2018

Academic Editor: Gerardo Pelosi

Copyright © 2018 Zhiyuan Jiang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

To quickly verify and fix vulnerabilities, it is necessary to judge the exploitability of the massive crash generated by the automated vulnerability mining tool. While the current manual analysis of the crash process is inefficient and time-consuming, the existing automated tools can only handle execute exceptions and some write exceptions but cannot handle common read exceptions. To address this problem, we propose a method of determining the exploitability based on the exception type suppression. This method enables the program to continue to execute until an exploitable exception is triggered. The method performs a symbolic replay of the crash sample, constructing and reusing data gadget, to bypass the complex exception, thereby improving the efficiency and accuracy of vulnerability exploitability analysis. The testing of typical CGC/RHG binary software shows that this method can automatically convert a crash that cannot be judged by existing analysis tools into a different crash type and judge the exploitability successfully.

1. Introduction

Software development is an error-prone job, as a vulnerability caused by the negligence of the developer will have serious consequences once exploited by an attacker. In the case of buffer overflow vulnerability, an attacker can gain access to sensitive information in the memory or intercept the control flow of the program. In addition, an attacker can implant malicious code by inputting data outside the boundary of the buffer or overlaying the key data area of a program, such as the function pointer or return address if the code does not have a logical check on the buffer boundary, which will cause tremendous loss.

With the popularity of information technology, the scale of software has increased and the logic has become more complicated. In addition, the object to be measured is usually binary closed-source software. To speed up the discovery, analysis, and repair of software vulnerabilities, researchers have developed many automatic vulnerability mining and detection tools [1], such as AFL [2], Peach [3], and Bestorm [4]. With these tools, researchers engaging in security can find software vulnerabilities with abnormal sample data without understanding the operating logic of the software being analyzed. For example, the researchers used a variety

of fuzzing techniques to generate approximately 20,000 crashes [5] in Microsoft Word. However, due to inadequate method for testing all these crashes, researchers had to divide them into 61 categories and submit them to Microsoft, and Microsoft found four high-risk vulnerabilities in these crashes. Unlike automatic vulnerability mining, vulnerability analysis as well as remediation requires a detailed understanding of the working mechanism and operational logic of the software being analyzed; thus, the implementation of repairs proceeds slowly and inefficiently. Obviously, the demand for a quick response in network space cannot be satisfied if we analyze the massive anomaly samples found by the vulnerability mining tool and implement the repair manually. Therefore, there is an urgent need to study the vulnerability assessment techniques, analyze massive amounts of abnormal samples, and quickly identify the vulnerabilities that are most likely to be exploited by attackers, to improve the pertinence of vulnerability repair [6].

At the present stage, there are two main categories of methods for evaluating the exploitability of binary software vulnerabilities: one is based on the model of the exploitability assessment method; the other is based on the use of exploit code generated by exploitability test methods.

Model-Based Exploitability Assessment Method. The exploitability estimation method refers to the method of collating and analyzing the existing vulnerabilities, establishing the feature models of exploitable vulnerabilities, designing the exploitability evaluation rules and strategies, evaluating the software status of the abnormal crashes according to these rules and strategies, and determining whether a given vulnerability is exploitable. Typical pattern-based analysis methods include !Exploitable [7], CrashWrangler [8], and CERT Triage Tools [9].

Exploit Code Generated by an Exploitability Test Method. The exploitability test method refers to the method of analyzing the state when the software crashes, the input file that caused the crash and the executable software code itself, and studying the construction of vulnerability exploitation code to automatically generate sample code that can prove the exploitability of vulnerabilities. Typical methods based on exploitability testing include Heelan AEG [10], MAYHEM [11], and Q [12].

Compared with the exploitability estimation method, the exploitability test method is more accurate, but the technology is still in the preliminary research stage--the scale of the object being analyzed must be small (the vast score can only analyze small-scale software), and the applicable exception sample data type is limited.

At present, the vast majority of vulnerabilities that are found by vulnerability mining tools are in the form of memory access exceptions [13]. The three major categories are read exceptions, write exceptions, and execution exceptions. At the present stage, both the pattern-based and the exploitability test methods can only be applied to the problem of the exploitability of write exceptions and execution exceptions [14]; these methods cannot determine the availability of the vulnerabilities that are caused by read exceptions.

Main Contribution. To solve this problem, the paper presents an exploitability analysis method based on automatic exception suppression on a binary program analysis platform named angr [15]. Our key contribution is performing a symbolic replay of the crash sample, constructing and using a self-referencing linked list or reusing data gadget, to bypass the complex exception, thereby improving the efficiency and accuracy of vulnerability exploitability analysis. Through the tracking and analysis of the implementation process of the vulnerability exception sample, the method can automatically generate new test cases that can suppress the read exception, according to the software execution path and the corresponding symbolic constraint condition, and convert the read exception with unknown exploitability into a write exception or a execute exception to complete the exploitability judgement. Experiments on 12 programs with read exceptions in CGC (Cyber Grand Challenge) [16] and RHG (Robot Hacking Game) [17] show that this method successfully constructs new test cases that can convert read abnormalities into execution or write exceptions and further convert crashes that cannot be judged or exploited by existing tools into exploitable crashes. The method can judge the exploitability of reading exception crash to reduce the false-negative rate of exploitability judgement effectively, thereby

making the exploitability judgements more reliable. Moreover, the automatic judging process greatly improves the efficiency of the exploitability analysis, which can quickly complete many crash exploitability judgements. The paper expands upon angr's Tracer functionality and rex's crash triaging and exploration.

The remainder of this paper is structured as follows. Section 2 introduces the preparatory knowledge and then elaborates on the general idea of the exploitability judgement of program vulnerability based on automatic exception suppression. Section 3 introduces the specific method of symbolic replay that is used in this technique. Section 4 introduces the method of automatic exception suppression. Section 5 presents the experiment and analysis, and the last section gives the conclusions.

2. Research Motivation

2.1. Read Exception Vulnerability. There are many types of vulnerabilities in software that are related to memory. Vulnerabilities detected by automining tools such as AFL are usually memory vulnerabilities that can cause application crashes. The types of crashes can be divided into three categories: read exceptions, write exceptions, and execute exceptions. For write and execute exceptions, there are a number of mature methods (such as function pointer tampering, ROP [18], and jumping to shellcode [19]) for achieving control flow hijacking or access to critical information. Once the exception samples cause the program to generate controllable write and execute exceptions, it is generally assumed that the vulnerability is exploitable and should be given priority in repair. Existing vulnerability analysis tools, such as MAYHEM, Q, PolyAEG [20], and REX [21], can automatically generate the exploit code for write and execute exceptions. These tools are based on symbolic execution [22] technology. They automatically construct input data that can accurately cover the function pointer or return address to achieve program control flow hijacking to obtain proof of exploitation.

However, in reality, many of the highly exploitable vulnerabilities are often difficult to arrive the execute exception or write exception directly after the exploitable vulnerability is activated, but first to the read exception. There are some examples that show actual exploits can be obtained from read exception from real-world scenarios: they are CVE-2011-0065, CVE-2013-3897, and CVE-2016-9079; there are also actual exploits obtained from write exception from real-world scenarios: they are CVE -2010-3333 and CVE -2011-0104.

Sample Program 1 is a program that has a stack buffer overflow. The program uses the read function to read data from standard input to a 128-byte buffer and reference the data in the buffer in subsequent code. The memory map after the program runs is shown in Figure 1. Since *buf* has only 128 bytes, more than 128 bytes ("AAAA") will overwrite the *tmp* pointer behind *buf* when inputting 132 bytes of "A" data from the standard input, to change it from "Hello world" to point to "0x41414141". Since the memory area (0x41414141) is not mapped by the memory management system, reading the area will cause memory read exceptions.

```

(1) char *message = "Hello World!";
(2) void vulnerable_function(){
(3)     char *tmp;
(4)     tmp = message;
(5)     char buff[128];
(6)     read(0,buff,256);
(7)     printf("%s",tmp); //The location read exception occur
(8)     printf("You have passed successfully!");
(9) }

```

PROGRAM 1: A simple program containing read exception vulnerability.

FIGURE 1: (a) The memory map of program with 128-byte external input. (b) The memory map of program with 150 “A” as external input.

According to the code, as long as the input data are long enough, the input data can not only cover the *tmp* pointer but also override the return address to hijack the control flow. However, since the program triggers a read exception before executing the return address, it crashes before the control flow is hijacked. For the exploitation of such a crash by a read exception, the exploitability tools usually return “unexploitable” or “unknown”.

The Shellphish team at the University of California, Santa Barbara, was concerned about this issue and tried to analyze the exploitation of the read exception. They developed the REX [15] vulnerability analysis tool based on angr [23] platform. REX regards the crash caused by the read exception as “exploitable” and further explores and transforms the test case to convert to the write exception or execute exceptions [24]. REX currently only explores a single read exception and cannot be applied to programs that contain multiple read exceptions. REX replaces the invalid address by using a mapped memory address that is controlled by external input to suppress the read exception; however, for less or noncontrollable memory, REX cannot be applied. In addition, REX is currently unable to address programs on the Linux platform, especially when the crash occurs in an external function.

In view of the above problems, we will introduce the corresponding solution and implementation methods in this paper.

2.2. The Main Solution. Since the exploitation of a crash of type write/execute exception is easier to judge, if the read exception can be suppressed, we do not let the binary program crash in the place where the read exception occurred; instead, we let it continue until it triggers the execution/write exception. At this time, the problem of judging the exploitability of a read exception is transformed into a problem of judging execution/write exceptions. The problem is greatly simplified and can be solved using the proposed method, in combination with the existing exploitability judgement method.

For example, for Program 1, the overflow occurred due to the *tmp* pointer to the memory area being invalid, which resulted in *printf* causing a memory read exception. By analyzing and tracking the state of the memory during program execution, we find that the data covering *tmp* come from external data and *tmp* is controllable. If we overwrite *tmp* as a memory address that has been mapped by the memory management system, when the *printf* function calls *tmp*, it will not cause read exceptions, and the program can continue to execute until reaching the return function. As long as the external input is long enough, the buffer overflow will continue to overwrite the return address into an invalid memory area, where the execution of the program will occur, as shown in Figure 2. If the program encounters another read exception, we can use the same method to suppress it until all read exceptions have been eliminated by the suppression

FIGURE 2: (a) The memory layout of program with 150 “A” as external input. (b) The memory layout of program with constructed data that can suppress read exception as external input.

FIGURE 3: Main framework of our method.

FIGURE 4: Flow chart of symbolic replay.

program, or an exception is encountered. In this way, we convert the complex read exception exploitability judgement problem into a simple write/execute exception exploitation judgement problem successfully.

In this process, the most important questions are how to automatically determine which memory areas are valid (mapped) when detecting read exceptions and how to automatically generate new test cases that can mitigate read exceptions based on these valid (mapped) memory areas.

To this end, we design and implement an automatic exploitation analysis system based on exception conversion; the main framework is shown in Figure 3. The input to the system is a sample of data that can cause the test program to crash, and the system collects all symbolization constraints of the crash moment in a symbolic replay link. If the type of the crash is a read exception, the read exception is suppressed and a new test case can be generated using the symbolic solution to suppress the current read exception. The new test case is reconverted as an input to the automatic exploitation analysis system until the conversion is successful (discovery of an execute/write exception) or the symbolic constraint has no solution, in which case the conversion fails.

3. Symbolic Replay

Symbolic replay is mainly used to track the control flow and data flow after the abnormal sample data are input into the analysis program, to obtain the path constraint condition and the symbolic memory information when the program crashes, and then provide the basic information for the subsequent read exception suppression. For example, we use concrete value as input when given a program and run it until crash, we obtain the program trace during the process. After that, we run the program again, but we use symbolic value as input this time. The program will finally crash under the guidance of program trace, and we can obtain symbolic constraints at crash moment. Symbolic replay is divided into two phases: recording the real execution of the trace and obtaining symbolic constraints. The flow char of symbolic replay is shown in Figure 4.

The input of the entire symbolic replay process is a test case that can cause a program to crash, which is obtained from a vulnerability mining tool or by other channels. First, we perform the concrete execution of the target program. This step has two purposes: (1) to ensure that the external input

Input: Trace list t logged by real execution
Output: None

- (1) $A_{\text{current}} :=$ current address of symbolic execution
- (2) $S :=$ step size of symbolic execution
- (3) $Bb :=$ the count of basic block, $Bb = 0$
- (4) load the target binary
- (5) symbolically execute the binary until $A_{\text{current}} == t[0]$
- (6) **while** $A_{\text{current}} == t[Bb]$
- (7) $Bb = Bb + 1$
- (8) $S = t[Bb] - t[Bb - 1]$
- (9) $A_{\text{current}} =$ the address after symbolic execution for step size S
- (10) if $Bb \geq \text{len}(t)$ or crash occurred
- (11) return crash state
- (12) **if** $A_{\text{current}} != t[Bb]$
- (13) call Path_correct()
- (14) **End**

ALGORITHM 1: Supervised symbolic execution.

can make the program crash and (2) to obtain the concrete execution of the trace as a guide for the follow-up symbolic execution. After the completion of the concrete execution, restart the program and begin symbolic execution under the guidance of the concrete execution of the trace until the crash. Then, obtain the symbolic constraints of the crash moment. The following is a description of the symbolic replay in three subsections. The first section is the use of symbolic execution in the process of symbolic replay, the second section is the proposed selective simulated execution whose purpose is to speed up symbolic execution, and the last section is an automatic correction method for path divergence occurring in guided symbolic execution.

3.1. Symbolic Execution under the Guidance of the Concrete Execution of the Trace. There are many exception types that can cause a program to crash, but most of the exploitable exception types are invalid memory references and illegal instructions, and all read exceptions fall into one of these two types. Therefore, we only consider the two types of crashes for exploitation judgements. To determine the type of crash in the shortest possible time, we use QEMU [22] to concretely execute the program. Instrument is performed when concretely executing the program to record the trace of the program's implementation from the beginning to the crash. We use the Linux signal to determine the type of crash. If the return value of signal is SIGSEGV or SIGILL, it is a treatable exception; otherwise, we have encountered an exception type that we cannot handle, and the process will automatically exit and prompt the external input trigger exception whose type cannot be processed.

3.1.1. Obtaining the Concrete Execution of the Trace. The trace of the guide must be sufficiently long to reflect the execution path of the program. The basic block is a sequence of statements in a program execution process, with only one entry and one exit. Therefore, as long as we record each basic block at the beginning of the address in the

process of execution, the trace can objectively represent the implementation of the program path. Trace record in units of basic blocks is called the coarse-grained record, and the trace record in units of instructions is called the fine-grained record. The trace in the coarse-grained record is thinner, but it is not possible to inspect the external function (like libc functions, including printf () and read ()); system function (referring to system call functions, including fork () and exit ()). Only when the program crash position is in an external function or system function can we use fine-grained records to record the external/system function of the crash, and the rest of the function still uses coarse-grained records to ensure that the trace is sufficiently compact.

3.1.2. Guided Symbolic Execution. Symbolic execution, in simple terms, is the use of a symbolic value instead of the real value to execute the application, by converting the predicate condition of the application execution trace into a logical expression and a path constraint expression. Usually, the contents that need to be represented symbolically in the symbolic execution process are stdin, file, network input and output, the command line, and so on. Since these input data are represented as symbolic variables, it is difficult to determine the path to be executed in the next step when the program executes a jump instruction that depends on the symbolic variable. To ensure that we can effectively collect the path constraints and symbolic memory information in the event of an exception, we need to use the concrete execution of the trace to monitor the symbolic execution, to guide its choice in the symbolic jump to ensure that the symbolic execution can arrive at the crash position. The "guide symbolic execution" relies on features of angr, but it is weak in coupling. Therefore, it is feasible to use other tools if they can provide better functions.

The algorithm for performing symbolic execution based on the guidance of the trace is shown in Algorithm 1. Since the initial states of symbolic execution and concrete execution are different, it is first necessary to align the symbolic execution


```

class printf(FormatParser):
    def run(self):
        fmt_str = self._parse(0)
        out_str = fmt_str.replace(1, self.arg)
        self.state.posix.write(1, out_str, out_str.size()/8)
        return out_str.size()/8

```

PROGRAM 2: Summary of printf.

FIGURE 5: The judgement flow chart of simulated execution.

with the concrete execution trace so that the starting address of the symbolic execution is the same as the first address of the concrete execution trace. Then, begin the guided symbolic execution under the monitoring of the concrete execution trace and constantly update the path constraints in the implementation of the path. Bb represents the basic block number to be executed. The size of each step is S, which is equal to the difference between the next basic block address $t_{[Bb]}$ and the current basic block address $t_{[Bb-1]}$. Each time the symbolic execution finishes, the current address is updated with $A_{current}$, and if $A_{current}$ is the same as the address of the corresponding trace sequence, then Bb is incremented by one. Use this method for execution until reaching the last record of the trace or until the symbolic execution catches the exception.

3.2. Selective Simulated Execution. The speed of the real CPU is approximately 800,000 times faster than symbolic execution implemented in angr after optimization [25]; we should minimize the number of instructions that need to be symbolically executed. For the objects that can be reduced, the general binary program includes the main program, custom external library functions, and system library functions. Among them, the main program and the custom external library contain code that is closely related to the program's own operating logic and are versatile while system library functions are universal; thus, system library functions are the main objects of our reduction. For example, the commonly used *printf* function, which belongs to system library *libc*, is very complex, containing several function calls. A program is often accompanied by a large number of system functions. If all of the instructions are symbolically executed, the analysis and processing time will be greatly increased,

which will result in very low operating efficiency. Therefore, it is necessary to adopt a method for avoiding the repetition of these large-scale library functions.

Our approach is to use selective simulation execution techniques, that is, to replace the real function with a shorter, functionally identical function summary. For example, a summary of the *printf* function can be represented as in Program 2.

When the symbolic execution engine detects that the current execution address is in the plt table, it indicates that the external library function is to be called. Then, the code is executed in the function summary to simulate the function and the constraints of the function. The whole process is shown in Figure 5.

By using simulation execution, we can significantly reduce the number of basic blocks that require symbolic execution. Statistically, the use of simulation execution can reduce the need for symbolic execution of a basic block by more than 60%; the more external system library functions that are called, the more obvious this effect.

The advantage of simulation execution is that it allows us to avoid dealing with complex external functions; the disadvantage is that if a crash occurs in an external function, because there is no real execution of these functions, we cannot perform crash analysis and processing. To handle a crash caused by the readout of an external function, we used selective simulation execution; that is, we real execute the external function and system function where exceptions arise, while performing simulate execution for functions in which no exceptions occur. The method is shown in Figure 6.

Before starting the symbolic execution, hook all external functions and system calls; when calling external functions and system functions, the program will automatically execute

```

(1) Input: real trace  $t$ 
(2) Output: None
(3) while  $A_{\text{current}} \neq t[Bb]$ :
(4)   If  $\text{is\_hooked}(\text{current\_addr})$ 
(5)     while  $\text{addr\_in\_plt}(t[Bb])$ :
(6)        $Bb += 1$ 
(7)      $S = t[Bb] - t_{\text{diff}}$ 
(8)      $A_{\text{current}} = \text{the address after symbolic execution with step size } S$ 
(9)   endwhile
(10) End

```

ALGORITHM 2: Make symbolic trace t_s consistent with real trace t .

FIGURE 6: The judgement flow chart of selective simulated execution.

our simulation function. Next, trigger the vulnerability, collect the information about the crash scene, and determine the location of the crash. If the crash occurs outside the program (i.e., the crash occurs in the external function and the system call), we unhook the function in which the crash occurs and then reexecute the program. The new execution will concretely execute the external function/system call in which the crash occurred and obtain symbolic constraints from outside the program.

3.3. Automatic Correction of Path Divergence. In the process of guiding the symbolic execution using the trace logged by concrete execution, some external functions and system functions are not really executed in the symbolic execution because of the implementation of the simulation functions, so the speed of the symbolic execution does not coincide with the concrete basic block velocity (usually, symbolic execution is faster). Therefore, when calling the system/external function, sometimes the path obtained by guided symbolic

execution is different from the path of the trace; we call this path divergence. To ensure that the trace that is logged by concrete execution can guide symbolic execution effectively, when path divergence occurs, we need to automatically correct it. Our approach is in Algorithm 2.

When the path of symbolic execution and the path of the trace are inconsistent, we first judge whether the current function is hooked; if it is hooked, this indicates that the path divergence is caused by a difference in speed between symbolic execution and concrete execution. Then, we continue to determine whether the corresponding trace of the current Bb (basic block) is in the plt (Procedure Linkage Table, PLT). If it is in the plt, which means that the trace is encountering external functions, the symbolic implementation is running fast at that moment. We need to constantly update the corresponding basic block of the trace until it catches up with the symbolic execution. We continue to increment Bb by 1 until the real execution of the trace has exited the external function and returned to the binary

FIGURE 7: Automatic correction of path divergence. (The trace of concrete execution is on the left; the trace of symbolic execution is on the right.)

internal. We let the symbolic execution step S instructions occur before the divergence address, where S is the difference between the current address $t[Bb]$ and divergent address t_{diff} , as shown in Figure 7.

Thus, we can make the symbolic execution and the concrete execution consistent. The guided symbolic execution will finally arrive at the point at which the crash occurred according to the trace logged by concrete execution. At this point, we have completed the task of symbolic replay.

4. Automatic Read Exception Suppression

4.1. Option for Automatic Judgement of Read Exceptions. In the concrete execution link, we determine whether the type of crash can be processed based on the Linux signal mechanism. However, the Linux signal mechanism is based on a relatively rough classification. For instance, SIGSEGV can be caused by a read exception or by an execution exception. Thus, we need to collect the symbolic memory information by symbolic replay to complete the judgement of exception type.

Symbolic memory refers to memory in which the memory values are represented by symbolic expressions and are affected by the input data. Instructions that directly lead to crashes or the last instruction in the concrete execution are called abnormal instructions; address used in abnormal instructions that cause exception is called violation address. Vulnerabilities can be divided into three types based on the analysis of the state of the program when the exception occurs, namely, execution exceptions, write exceptions, and read exceptions, as follows:

- (i) If IP (Instruction Pointer) or BP (Base Pointer) is symbolic when program crashes, the vulnerability is an execution exception.

- (ii) If IP and BP are not being symbolic, there are operations involving symbolic memory in the symbolic trace.
- (iii) If the operation is a write operation, the vulnerability is a write exception.
- (iv) If the operation is a read operation, the vulnerability is a read exception.

If the crash is at a nonsymbolic address, it means that we cannot control the address used in the crashed instruction by constructing the input and that there is no opportunity to suppress the exception.

Before introducing the suppression method for read exceptions, the definitions of valid, invalid, and violation addresses must be clarified.

The valid address refers to an address area being mapped by the operating system's memory management system, to which the program has access rights. The valid address is easy to find: it is the address where the main program is loaded into the memory and controlled by users in the symbolic memory. The invalid address is not mapped or the program does not have permission to access the address. Read, write, and execution exceptions will correspond to read, write, or execution operations at the invalid address. The violation address is the invalid address when an exception occurs. A memory read exception will occur when the program performs a memory read at an invalid violation address.

4.2. Simple Type of Read Exception Suppression. To suppress a read exception, we only need to ensure that the violation address is replaced with a valid address when the exception occurs. For a single read exception, that is, for an exception that is caused by a read operation at the violation address, the program can smoothly read a value from a valid address when

FIGURE 8: Suppressing exception by replacing violate address with valid address.

the violation address is replaced with a valid address; thus, a single read exception can be suppressed and the program can proceed. A diagram of this procedure is shown in Figure 8.

The explosion symbol indicates that the program crashes, yellow colour indicates that the crash type is a read exception, and red colour indicates that the crash type is an execution exception or a write exception. To automatically replace the violation address in the read exception with a valid address as well as obtain an external input that can suppress the current read exception at the next execution, we add a new symbolic constraint to the current symbolic constraint at the crash. The new constraint has a violation address that is the same as a valid address in the invalid address list. We traverse the list of valid addresses (valid address list = code segment address list + symbolic memory address list) until we find an address for which the above symbolic constraint is solvable and the symbolic solver can obtain a new input that satisfies the condition constraint. If the read exception occurs again during subsequent execution, use the same method to mitigate the exception until the program returns or an execution exception occurs. In Program 1, for example, a read exception occurs when the *printf* function dereferences the *tmp* pointer, as the *tmp* pointer is now overwritten as 41414141. Therefore, the current violation address is 0x41414141. Then, we add a symbolic constraint so that the violation address is equal to 0x7FF6770, which has been mapped by the memory management system. The new test case can be obtained by using the symbolic constraint, and the new test case is used as the external input of the program. The program reads the address 0x7FF6770 when the pointer *tmp* is dereferenced and no exception occurs. When the program executes the vulnerable_function *RET* instruction, since the *RET* is now overwritten to 41414141, the program meets

the execution exception. At this point, the conversion from reading exception to execution is complete.

4.3. Multilevel Read Exception Suppression. As we have discussed above, a valid address is needed when we try to suppress the current exception caused by an access violation. However, it will be quite difficult to get an address which is not only valid but also appropriate enough to ensure the suppression depth in the face of multilevel exception suppression. Two methods to manage to do that were proposed. If there is enough symbolic address control by the input and constraints are solvable, then we will build a data gadget. If not, we will look for an appropriate valid address inside the program automatically.

In modern software, with the extensive use of object-oriented technology, structures and so on, in many cases, a memory read exception is accompanied by the appearance of a multilevel pointer; we call a memory read exception that is caused by a multilevel pointer (or pointer of a pointer) a nested memory read exception. Compared with the single read exceptions discussed above, multilevel memory read exceptions are more complex and common. An example is given in Sample Program 3.

In example of Program 3, *Pstruct* is a pointer to the struct named *structArray*, and *structArray* contains a multilevel structure pointer. When the program overflows, the *Pstruct* pointer is overwritten by external data. If *Pstruct* is overridden as an address that is not mapped by the memory management system, an exception will occur when the program tries to dereference *Pstruct*.

To suppress the single read exception, we only need to ensure that the equation $\text{violate_addr} == \text{valid_addr}$ can be solved, because the valid address of the stored content does

```

(1) typedef struct my_struct{
(2)     int field1;
(3)     char* pMessage;
(4)     struct my_struct * pNext;
(5) } MY_STRUCT;
(6) typedef struct data_field{
(7)     int field;
(8)     char* message;
(9) } DATA_FIELD;
(10) void vulnerable_function(MY_STRUCT * structArray) {
(11) MY_STRUCT * pStruct = structArray;
(12) char buff[128];
(13) read(0,buff,256); // you can overflow pStruct here
(14) MY_STRUCT * pItem = NULL;
(15) for (pItem = pStruct; pItem != NULL; pItem = pItem->pNext){
(16)     printf("The message of %dth item is:%s\n",
(17)         pItem->field1, pItem->pMessage); }
(18) int _tmain(int argc, _TCHAR* argv[]){
(19)     MY_STRUCT structArray[3];
(20)     DATA_FIELD dataArray[3] = { { 1, "I'm struct1" },
(21)         { 2, "I'm struct2" }, { 3, "I'm struct3" } };
(22)     for (i = 0; i < 3; i++){
(23)         structArray[i].field1 = dataArray[i].field;
(24)         structArray[i].pMessage = dataArray[i].message;
(25)         if (i!=2)
(26)             structArray[i].pNext = &structArray[i + 1];
(27)         else
(28)             structArray[i].pNext = NULL; }
(29)     vulnerable_function(structArray);
(30)     return 0; }

```

PROGRAM 3: A program that can occur in multilevel read exceptions.

FIGURE 9: The memory layout when Program 3 is running.

not change the effect of read exception suppression. For the complex read exception represented by Program 3, the value read from the valid address has a direct or indirect effect on the next memory read operation. If the read exception occurs

with a continuous dereference, as shown in Figure 9, the value obtained by this dereference will be used as the address for the next dereference. Therefore, we need to ensure not only that the current memory read is valid but also that the read

0x7FFF3450	0x7FFF3450
0x7FFF3454	0x7FFF3454
0x7FFF3458	0x7FFF3458
0x7FFF345C	0x7FFF345C

FIGURE 10: The constructed continuously self-referential multilevel linked List.

content used as the memory address for reading memory next time is a valid memory address. Using this approach, no exception occurs in the following steps.

To solve this problem, we can divide it into two situations:

- (1) Symbolic memory that can be controlled by an attacker when a crash occurs (readable and writable)
- (2) No symbolic memory (readable and writable) that can be controlled by an attacker.

In the first situation, we can automatically construct an self-reference list. In the second situation, we need to search for data gadgets in memory space and reuse them.

4.3.1. Constructing a Continuously Self-Referential Multilevel Linked List. In practice, we use a self-referential list to suppress a read exception if user-controlled memory exists, as shown in Figure 10.

When read exception occurs, replacing violation address with address of self-referential list, program will do read operation to self-referential list. When dereferencing a multilevel pointer, as the data read from the specific memory address are a pointer to their own effective memory address, a solution must exist, no matter how many ranks the pointer to be dereferenced has. The exception caused by continuous deference can be suppressed by using one self-referential list, the exception caused by discontinuous deference can still be suppressed by using a series of self-referential list whose address are continuous.

As Figure 9 shows, pStruct contains three-structure pointer that are not continuously pointed. Data-struct1 whose address is 0x7FFF6684 is pointed first when dereferencing pStruct, while the value got from dereferencing is not the address of Ptr-struct1. Similarly, the value got by program is not the address of Ptr-struct2 when dereferencing Ptr-struct1. In order to suppress the read exceptions that caused by multi-illegal read operation with offset, we replace the violate address with the start address of constructed self-referential list. For example, when dereferencing pStruct that is overwritten with invalid address from external input in Program 3, read exception will happen. This exception can be suppressed by using the start address (0x7FFF3450) of constructed self-referential list shown in Figure 10 to replace the invalid address, because program can read a value (0x7FFF3450) successfully from the start address and then program steps into structure1. Ptr-struct1 will be dereferenced at the address relative to the entry address offset by 8 bytes in structure1; at this time, the address of Ptr-struct1 is $0x7FFF3450 + 8 = 0x7FFF3458$. The result of dereferencing Ptr-struct1 is 0x7FFF3458 since Ptr-struct1 is a self-referential

list now, then program steps into structure2. Similarly, Ptr-struct2 will be dereferenced at the address relative to the entry address offset by 8 bytes in structure2; Ptr-struct3 will be dereferenced at the address relative to the entry address offset by 8 bytes in structure3. There will not be any read exceptions when dereferencing pStruct because all of the addresses to be dereference are the address of self-referential list, which let the program continue to execute until ret.

By constructing this kind of continuously self-referential multilevel linked list, we can suppress (1) the read exceptions caused by dereferencing a multilevel pointer and (2) the read exception caused by dereferencing a multilevel pointer with offset.

4.3.2. Data Gadget Reusing. In the case of sufficient controllable symbolic memory or limited controllable symbolic memory, we construct the self-referencing multilevel list to achieve abnormal suppression. However, if no controllable memory cells can be found in the memory or the symbolic constraints defined above are not solvable under some extreme conditions, we can only discard the self-referential multilevel linked list that was constructed from the input controllable memory area. In such circumstances, we need to find multilevel linked list as data gadget in the program memory space.

In process of searching for existing linked lists, to ensure effectiveness, we first search for a linked list in the program code section and then search the external library and the list on the stack. However, through real software tests, we found that the ranks of linked lists in the code segment are low; large-scale programs are no exception. To *tcpdump* [26], for example, the highest rank of linked lists in the code segment is only 3, while the rank of the linked list in the stack where read exceptions usually occur is often much higher than 3. We find that the highest rank of the linked list in the stack is 7 when a crash occurs in *tcpdump*. We have to consider the worst situation, a memory read exception caused by the dereferencing of a multilevel pointer, while the rank of the most stable linked list from the code segment cannot meet the requirement; moreover, no address can be written, so an attacker cannot complete the multilevel dereference by constructing a self-referential linked list. With available tools, such as PED4 [27], one can quickly and accurately collect the linked list as the program runs. We will first create a list of the pending linked lists with linked lists that we have collected and sort them in descending order based their ranks. Then, we need to filter this list to obtain a data gadget that can be used to replace the violation address. The selected data gadget needs to have the following characteristics: (1) the data gadget address is in the solvable range of the symbolic pointer that points to the place where the program crashes. (2) The violation address can be made equal to the data gadget address by constructing an external input.

As shown in Algorithm 3, we first determine the symbolic range of the violation address, which is the address range that can be pointed to when the program crashes, leaving only the linked list that can be pointed to at the time of the crash. Next, we add a new constraint to the original constraint of the program: the violation address must be equal to the address

```

(1) Input: Original chain list L
(2) Output: None
(3) data_gadget_list = [];
(4) min_read = solve_constraints.min(violating_action_addr);
(5) max_read = solve_constraints.max(violating_action_addr);
(6) filtered_chain = filter(lambda x: (min_read <= x) and (x <= max_read), largest_regions);
(7) for chain in filtered_chain;
(8)     new_constraint = violation_action_addr == chain;
(9)     if solve_constraints.satisfiable(previous_constraint + new_constraint)
(10)         data_gadget_list.append(chain);
(11) End

```

ALGORITHM 3: Filter original chain to get data gadget.

of the linked list, which ensures that the linked list address is within the solvable range of the symbolic pointer. Then, we use the constraint solver to try to solve all the constraints, leaving only the list of those that can be solved as a data gadget. The alternative data gadget linked list formed through these two steps is what we need.

Since the contents of the stack change dynamically, which means different processes will have different stack addresses, and ASLR [26] is very popular in program development, we cannot guarantee the existence of stable stack addresses that is available for replacing violation addresses. The head address of the linked list in the stack that was found by the data gadget search, whose rank is sufficiently high, can be hard-coded into the symbolic constraint, which will lead to a breakthrough failure out of stack address failure in the next execution round. To solve this problem, we refer to the jump to register method used in exploit technique [28] concept of vulnerability exploitation to replace the original method of hard-coding the address into the symbolic constraint and use the register address plus the offset address as the substitutional address. The reason why the substitution option works is that the offset of the searched valid linked list relative to the register remains unchanged when a certain instruction is executed, although the stack address changes dynamically as execution proceeds. In Program 3, for example, at the beginning of the crashing block, we found that there is a three-level linked list relative to the top of the stack 0x24 offset, and the offset is not affected by the program process. Therefore, our new symbolic constraint expression is $violate_addr = esp + 0x24$, so we can always make the violation address point to the four-level linked list, no matter how the stack address will change, thereby guaranteeing that the read exception caused by the dereferencing of the structure that contains multilevel structure pointers can be suppressed.

The method above has an obvious effect on most complex read exceptions, but there are still some programs that cannot be effectively suppressed. Such programs have the following characteristics: they contain more read exceptions (usually more than 10), read exceptions occur at different addresses, the addresses of exception are far from one another, and the logical relationship between them is not obvious. For such programs, the read exception suppression for each level is

affected by the contents read by the upper-level suppression: the contents of the previous read affect the symbolic range of the current dereferenced address. If the suppression on the previous level of the read exception is not appropriate for using the data gadget, it will cause the subsequent read exception to become difficult to suppress due to increasingly complex constraints, which will reduce the suppression level of the read exception. Obviously, the more the read exception level can be suppressed, the more likely it is to convert to other exceptions. In the method described above, the suppression of the read exception is implemented by selecting a data gadget to replace the original violation address from the optional data gadget list, and the selection of the data gadget is random. This random selection of data gadgets does not take full account of the impact of data gadgets on program execution and cannot guarantee higher levels of suppression. Therefore, we will convert the selection of the data gadget into a search problem, taking full account of the effect of the read exception suppression link on subsequent program execution and proposing a selection strategy for the data gadget that maximizes the level of suppression.

In the search process, we regard the instruction that causes the exception as an exception node. Exception nodes are divided into three types according to the exception type: read exception nodes, write exception nodes, and execute exception nodes. Each exception node maintains a list of available data gadgets of length L and its own level, and the level of exception nodes indicates the number of exceptions that the program triggers when the program is executed on that node, as shown in Figure 11.

When the exception node succeeds in selecting a data gadget from the list of available data gadgets to replace the violation address, the exception node becomes a suppressed node. A suppressed nodes have development attributes, which indicates that the program continues executing from the suppressed node up to the final state that the suppressed node can reach. The development attributes are normal exit and new exception nodes. The new exception node level is equal to the suppressed node's level plus one. For a suppressed node with level n , if the development attribute is a normal exit, all of the exceptions on the current execution path have been suppressed and there is no possibility of an exception type conversion, so the data gadget used by the suppressed

FIGURE 11: Each exception node maintaining its own level and available data gadget list.

FIGURE 12: The diagram of searching data gadget group.

node is removed from the data gadget list. For a suppressed node with level n , if the development attribute is a new exception node, but the new exception node's available data gadget length is 0, the new exception node cannot find a data gadget that can be used to replace the violation address, which means the current exception node cannot be suppressed. Therefore, we remove the data gadget used by the level n suppressed node, which causes the level $n+1$ exception node's suppression failed to be removed from the list of available data gadgets maintained by the level n suppression node. Each time a new exception node is found; its exception type is determined. If it is of write/execute exception type, the exception conversion is completed and the search is finished. If it is still of read exception type, the search is continued.

Figure 12 shows a simple search example. For an exception node whose type is read exception with a level of n , there are eight alternative data gadgets in the available data gadget list. First, use the first data gadget in the list to replace the violation address; the data gadget group is $\{g_1\}$, the current exception node becomes the suppressed node, and the development attribute is the new exception node, whose type is read exception. However, the available data gadget list of the new read exception node is empty. Therefore, go back to

the level n suppressed node, remove data gadget g_1 and use g_2 to explore. After using g_2 , the level $n+1$ read exception node successfully becomes the suppressed node, the data gadget group is $\{g_2, g_1\}$, and its development attribute is finding new read exception node, whose level is $n+2$. However, by traversing the available data gadget list $\{g_2, g_1/g_2/g_3/g_4\}$ of the level $n+1$ node that was explored with g_2 , we did not find a data gadget that could be used to suppress the level $n+2$ exception node. At this point, return to the level n suppressed node, delete g_2 , and use g_3 to reexplore. By using the data gadget group $\{g_3, g_1, g_2\}$, the level $n+2$ exception node is suppressed, and the development attributes of the current suppressed node results in finding a new exception node; the exception type of newly found exception node is execution exception, which means that exploration has been completed and conversion of the exception type has been achieved.

5. Experiment and Analysis

The DARPA Challenge Binaries (CBs) are custom-made programs specifically designed to contain vulnerabilities that represent a wide variety of crashing software flaws. They are more than simple test cases, they approximate real software

TABLE 1: Experimental results.

Program name	Exception type	Conversion method	Conversion count	Converted type
CADET_00001	Read exception	Gadget	1	Execute
CROMU_00055	Read exception	Self-reference/gadget	4	Execute/write-x-where
CROMU_00058	Read exception	Self-reference/gadget	1	Execute/write-x-where
CROMU_00064	Read exception	Self-reference/gadget	12	Execute
CROMU_00078	Read exception	Gadget	5	write-x-where
YAN01_00001	write-x-where	Gadget	8	Execute
LEGIT_00003	write-x-where	Self-reference/gadget	1	Execute
TNETS_00002	Read exception	Gadget	5	Execute
NRFIN_00074	Read exception	Self-reference/gadget	3	Execute
NRFIN_00078	Read exception	Gadget	2	Execute
CROMU_00064	write-x-where	Gadget	4	Execute
Equation_parser_overflow	Read exception	Self-reference/gadget	1	Execute

TABLE 2: Comparison between our method and existing tools (“√”—Exploitable “×”—Probably Exploitable “x”—Unexploitable “?”—Unkonwn).

Program Name	Exception type	!exploitable	CERT Triage Tools	CRAX	REX	Our method
CADET_00001	Read exception	?	√	×	×	√
CROMU_00055	Read exception	?	?	×	×	√
CROMU_00058	Read exception	?	×	×	×	√
CROMU_00064	Read exception	√	√	×	×	√
CROMU_00078	Read exception	?	?	×	×	√
YAN01_00001	write-x-where	?	?	×	×	√
LEGIT_00003	write-x-where	?	√	×	√	√
TNETS_00002	Read exception	?	?	×	×	√
NRFIN_00074	Read exception	?	?	×	×	√
NRFIN_00078	Read exception	?	?	×	×	√
CROMU_00064	write-x-where	?	?	×	×	√
Equation_parser_overflow	Read exception	?	√	×	×	√
NRFIN_00059	Read exception	?	?	×	×	×
KPRCA_00065	Read exception	?	?	×	×	×

with enough complexity to stress both manual and automated vulnerability discovery. The CBs come with extensive functionality tests and triggers for introduced bugs, patches, and performance monitoring tools, enabling benchmarking of patching tools and bug mitigation strategies [29]. CGC binaries are supposed to reflect binary verification challenges in the real world. The scales of these challenge programs range from hundreds of lines of C code to nearly a million lines. There are multiple challenge binaries with bugs that mimic real-world vulnerabilities found in the past in software like sendmail and MSSQL [30]. We selected some CGC and RHG programs with read exceptions to study the ability to judge exploitability based on the automatic exception suppression method. The experimental results show that our method not only is effective for read and write exception crashes but also has a better conversion effect than other approaches. By suppressing the write exception, we can also find an execution exception. Exception Type in Table 1 represents the type of program crash caused by the initial test case,

and Convert Method represents the method used to suppress the exception, including constructing a self-referential linked list (self-reference) and using the data gadgets. Conversion Count indicates the number of different addresses encountered by the program for which the exception type is the same during the process from suppressing the first exception to finishing exception conversion. Converted Type indicates the type of crash triggered by the new test case after the conversion is complete: execute exception (Execute) and write exception (write-x-where). The conversion experiments were conducted on a personal computer equipped with an Intel Core i7-5280k 3.3 GHZ CPU with 96 G of memory which ran Ubuntu 16.04. We only select the most representative of the procedures for analysis and discussion.

Table 2 shows the comparison between our method and existing tools, including: ! exploitable, CERT Triage Tools, CRAX, REX. The program in CGC was originally developed for DECREE, a custom Linux-derived operating system. In order to make them compatible with other platform, we use

cb-multios [31] to modify the original CGC binaries to work on Linux and windows. The result of check experiment shows that our method is more effective on exploitability judgement.

There are two showcases that failed to identify the exploitability of the binary and that even were exploitable according to the results of CGC [32] with our method. We provided the input generated by AFL, causing NRFIN_00059 and KPRCA_00065 read exceptions. And there was no exploitable bug found in neither NRFIN_00059 nor KPRCA_00065 by our method within sufficient time. After analyzing the results, we found that the following reasons resulted in the failure. The prerequisite of our method work is that there is at least one execution exception in the binary. However the exception in NRFIN_00059 was a memory-leak bug which we did not consider. In KPRCA_00065, AFL may generate a crash raising execution exception. But we found that execution exception happened in another condition branch and earlier than the read exception provided by our method. Although our method can explore different branches of the program backwardly, the depth of the backward searching is limited. The terminal point of the search is the initial read exception point. If the branch in which terminal point is located is different from that in which the crash point of execute exception is located, we cannot transit the exception successfully. We will try to improve the algorithm in next work.

5.1. Simple Read Exception Conversion. CADET_00001 is a program from CGC (CQE link) for which the difficulty level is hard. This program allocates 64 bytes of memory for external input but can read up to 128 bytes at a time, so there is a stack overflow vulnerability. The test case generated by AFL, which is of length 128 bytes, can lead to the program crashing. By using the test case as an external input, the program triggers a read exception at address 0x8048364, whose instructions are as follows: movsx ecx, byte ptr [ebp + ecx * 1-0x54]. Since ebp + ecx * 1-0x54 points to an address that is not mapped, a read exception occurs when program reads from this address. The violation address is $ebp + ecx * 1-0x54 = 0xfbebf0d5$. As the program is very complex, it is difficult to determine directly what leads $ebp + ecx * 1-0x54$ to point to the invalid address. Both !exploitable and REX determine that the crash is not exploitable.

In Figure 13, the red command indicates an exception instruction, the letter in the red circle before the instruction indicates the exception type, R is the read exception, and E is the execution exception. The green instruction indicates the first instruction of the newly arrived basic block after the exception has been suppressed. The number in the preceding green circle indicates the order of the instruction that the program has reached: 1 indicates the instruction at which the program arrives first after the exception has been suppressed. By analyzing the symbolic information that was collected during the symbolic replay link, it is found that the program crash is of read exception type, and our method automatically enters the exception conversion session. ebp and ecx are symbolic when program crashed, and we find that the symbol range of $ebp + ecx * 1-0x54$ is $0x1 \sim 0xffffffff$ at this time. The address of the program code segment

is 0x8048000~0x805cf65. There are two symbolic memory addresses: 0xbaaaaf4c and 0xbaaaaf98. We first select the code segment address to replace the violation address and found that the constraints are not solvable. Then, we replaced it with the symbolic memory address, but it still could not be solved. Therefore, we can only find the data gadget that can be used to replace the violation address. We found the solvable data gadget at the offset of 0x32, where the data gadget address is 0xbaaaaf4e. Solve the symbolic constraint to obtain a new test case and use the new test case as the program input. The read exception is successfully suppressed and the program executes to 0x80483ff (red circle E). Then, since the ret was overwritten by external input due to stack overflow, an execution exception is triggered and exception conversion is complete. Since the instruction pointer is completely controllable at this time, the crash is exploitable. Automating the conversion process takes a total of 42 seconds.

There had been massive cases with no results before our method found an exploitable bug finally. And thus, we did not observe every “no results” case since they are meaningless for our purpose, but just leave a parameter act as the threshold. When time consumption is over the threshold, the process of our method will be terminated, which means that it fails to find an exploitable bug. The thresholds of time consumption were 5 minutes for CGC binaries and 3 minutes for RHG binaries in our experiments. The threshold of time consumption is related to the program scale. It is clear that the larger the program scale is, the higher the threshold of time consumption should be. We can adjust the threshold according to the requirements. All of successful test cases in our experiments make it before reaching our threshold of time consumption.

5.2. Complex Read Exception Conversion. CROMU_00055 is a program from CGC (CFE link) for which the difficulty level is hard [33]. When the program receives an external input, a function called CallocAndRead() is called. This function applies a value-sized memory area for external input data, where the value is an external input, and the data are copied into the buffer. To ensure that all the strings are copied, this function requests a buffer with size equal to the external input data length plus 1 for these data. However, due to an error in the function definition, the buffer length parameter is passed as an 8-bit unsigned integer. Therefore, if the value is 255, +1 in the application of the buffer will lead to 8-bit integer overflow, thereby resulting in the allocation of too little memory for the length of the next read, which causes heap overflow. At 0x804ce57, the program triggers a read exception due to heap overflow, with the following exception instructions: move ax, DWORD PTR [eax]. At this time, the value of eax is 0xf1f1f1f7, which is an invalid address. The symbolic range of eax is $0x1 \sim 0xffffffff$ when the program crashes, the code segment range is 0x8048000~0x80659d3, and the symbolic memory size is 22. We try to use the code segment address and the symbolic memory address to replace the violation address and find that the constraint is not solvable. Then, we try to use a data gadget to replace the violation address to suppress the exception. Table 3 is the result of suppression from 0x804ce57.

TABLE 3: Exception nodes in suppression process.

Node series	Node address	Node type
(1)	0x804ce57	Read violation
(2)	0x804ca79	Read violation
(3)	0x804ca85	Read violation
(4)	0x804d08d	Read violation
(5)	0x804f710	Write violation

FIGURE 13: Suppression of CADET_00001.

TABLE 4: The conversion results of using different crashes.

Order	Crash_addr	Original_type	Convert_count	Final_addr	Final_type	Exploit-able
(1)	0x804f6b1	Read_violation	2	0x804f6cf	Exit	No
(2)	0x804ca79	Read_violation	7	0x804cdd8	Write violation	Yes
(3)	0x804fbee	Read_violation	4	0x804f710	Write violation	No
(4)	0x804fb1e	Read_violation	4	0x804cdf7	Write violation	No
(5)	0x804cb49	Read_violation	3	0x804cb00	Exit	No

FIGURE 14: Suppression of LEGIT_00003.

In the process of exploration, read exceptions are triggered at four different addresses. Through the data gadget search algorithm, we continue to choose the data gadget that can suppress the current exception and use the esp address as the data gadget to replace the violation address when read exception occurs at 0x804ce57; a series of read exceptions are suppressed after the address 0x804ce57. At 0x804f710, a new crash is triggered, which is of arbitrary write type, and the crash instruction is `mov byte ptr [ecx + eax * 1-0x1], 0x0`. The read exception at 0x804ce57 is successfully converted to a write exception at 0x804f710.

We use AFL to produce multiple different crash attempts for performing exception conversions to test the reliability of our method. The conversion results are shown in Table 4.

According to the conversion results, not all of the crashes can be converted. If there are no other exception types on the suppressed execution path, program will exit normally

after all of the read exceptions have been suppressed. The new test case obtained at this time is a legal external input (e.g., 1,5). Through debugging, we found that although new types of crashes can be found after exception conversion, they are not necessarily exploitable. In the third crash, for example, the exception type is converted from a read exception to a write exception, but the write address and write content are uncontrollable and therefore cannot be used.

5.3. Write Exception Conversion. LEGIT_00003 is a program from DEFCON 2016, which contains stack buffer overflow. By fuzzy testing of AFL, we obtain many test cases that can lead to crashing of the program. We choose one of the test cases that can cause the program to trigger a write exception.

In Figure 14, the red address indicates the location where the write exception occurred, and the yellow address indicates where the execution exception occurred after the write exception was suppressed. The program calls the function

with a vulnerability at 0x804832B. This function compares the values of $[ebp + var_10]$ and $[ebp + var_C]$ at 0x80481E6. If $[ebp + var_10] \geq [ebp + var_C]$, the program jumps to 0x80481F2 and updates the value of $[ebp + var_10]$. Until $[ebp + var_10] < [ebp + var_C]$, the vulnerability function returns 0x8048330 and continues. The write exception occurs at 0x8048201. The reason for the exception is that $ecx + eax$ points to an invalid address. The symbolic range of the write address is $0 \sim 0xffffffff$, and there are 7 sections of the controlled memory region when the program crashed. As long as the write operation at 0x8048201 does not cause an exception, the value of $[ebp + var_10]$ will be incremented by 1 every time the current instruction branch is executed, until $[ebp + var_10] < [ebp + var_C]$. By constructing a self-referential linked list/search data gadget, the violation address ($ecx + eax$) is made equal to the self-referencing list/data gadget address, and we can ensure that the program is executed smoothly to 0x8048330. Then, the program executes to 0x8048335 and triggers an execution exception, and exception conversion is complete. At this time, the return address is symbolic, the instruction pointer is controlled, and the current crash is exploitable.

5.4. Read Exception Conversion in Real Environment. Similar to CGC, RHG is a competition in which machines automatically perform vulnerability mining and exploitation; the difference is that RHG uses a real operating system environment (CentOS), while CGC uses a simplified operating system called DECREE, which has only seven system calls. In addition, RHG uses all the programs that are compiled in the real environment elf program, and the program does not have the symbolic table information. Compared to the CGC program running on the simplified operating system DECREE, the biggest challenge in completing the exception conversion in the RHG program is how to quickly and efficiently handle the impact of the real environment on the conversion process. As the real operating system contains many external library functions and system calls, symbolic execution technology conversion tools often cannot be used to correctly solve the many constraints that are generated by a real environment, which leads to exploitation judgement failure. Our method uses selective simulated execution and effectively solves the problem of path divergence due to simulated execution, so that we have the ability to automate the exploitation judgement of programs in real environments. Equation_parser_overflow is the program used in the online test match of RHG. The program has an overflow vulnerability, we can cause many crashes using AFL fuzzy test. Since the program is a program in a real environment, the existing automated analysis tools based on symbolic execution cannot automatically determine the exploitability. We apply our method for performing exploitation judgement to one of the test cases that can lead to a read exception; the experimental results are shown in Figure 15.

After executing 2913 basic blocks, the program triggers a read exception at the red circle marked with R, and, after using our method to suppress it, the program executes eight additional basic blocks and, in the 2922th base block, triggers an execution exception (the red circle marked E). For programs that contain this kind of read exception, due to

the occurrence of potential exceptions that are far away from one another, and the relationship between the exceptions is not obvious; thus, it is difficult to manually determine the exploitability of the program. By using our method to suppress the read exception, the execution exception is found automatically, which takes only 26 seconds.

In the RHG race, we (halfbit clan) used the automatic exploitation analysis system based on exception conversion to address the large number of crashes generated by AFL fuzzy testing for the first time. The results of the game prove that our method performs well in the RHG tournament in the same scenario: our robots can filter many crashes and successfully convert some complex read exception crashes into the write/execute exception crashes in a short time. With the help of this technology, our robots can generate pov (Proof of vulnerability) faster and eventually scored the first place in the RHG tournament with the highest offensive score [34].

5.5. Read Exception Conversion in Real Environment. In order to study the capability of fuzzing tools like AFL in our benchmark, we also did experiment to show their comparison. The results are shown in Table 5.

First AV(AFL) denotes the time that AFL consumes when finding the first access violation. First EV(AFL) represents the time that AFL consumes when it discovers the first execution violation. First converted EV (AFL+ method) denotes the time that our method consumes when converting the access violation found by AFL to execution violation. Besides, successful cases denote the number of inputs that our method converts the access violation found by AFL into execution violation. Fail cases denote the number of inputs that our method failed to. Convert time denotes the time consumed by a conversion process.

As observed from the experiment, among the 12 cases of violation conversion that our method completed, 8 cases could be found directly by AFL, and the 4 cases cannot be found by AFL in a certain period of time (1 hour). In the 8 cases, AFL are faster for 3 cases. After analyzing these programs, we found that the sizes of these binaries are small, and the location of vulnerability was shallow. When the vulnerabilities are deeper and the logic are more complex logic as in the 4 failed cases for AFL, it was very difficult for fuzzy test to generate the inputs that satisfy the corresponding conditions. As our method is based on symbolic execution, it could generate inputs that could reach these deep vulnerabilities by constraints solving.

AFL based on fuzzy test was very strong in finding potential vulnerability and the benchmark code that we used had relatively small scale, so most of the execution violation of our benchmark could be found directly through AFL as long as there was enough time. By contrast, our method was more targeted and more efficient for the vulnerability with more complex constraints and deeper codes. How to combine the advantages of our method with fuzzy test will be our research content in future.

6. Conclusions

We developed a binary software vulnerability exploitability analysis technique based on exception automatic suppression, which can transform crashes for which it is difficult to

FIGURE 15: Suppression of Equation_parser_overflow.

estimate the exploitability into a type for which it is easy to judge the exploitability. To complete the suppression from reading exception to writing/execution exception and writing exception to execution exception, we adopt the symbolic execution guidance, selective simulation execution, automatic correction of path divergences, and complicated automatic exception suppression methods based on the constructing self-reference linked list and reusing data gadget, thereby making the program suppress exception points and continue to execute until a new exception is triggered. This technique has a good conversion effect for read anomaly crashes in various situations and can be used to address various types of vulnerabilities, including stack overflow. We perform fuzzy tests on CGC and RHG programs by using the existing automation exploitability judgement tools to analyze and judge the crashes we have produced and to judge crashes that cannot be judged by other methods or have been judged to be

unexploitable. We successfully transform 12 crashes that have been identified to be unexploitable into exploitable crashes.

At present, this technology can only complete various exception type suppression for programs with many abnormal types. These kinds of suppression are achieved by suppressing the current exception to allow it to reach the next exception, so the premise of the successful suppression is that there must be at least one execution or writing exception after the current exception position of the program. Because the suppressing methods of this technique involve solving for a valid value using symbol to replace the conflict values, and the effect of suppression is also affected by the initial test case, in some cases, it will not transform into the new anomaly type but instead exits normally. In theory, the conversion success rate can be improved if we vary the test cases that cause the program to exit normally. In the next step, we will develop an exception conversion method based on exit test case

TABLE 5: Experiment results of our approach and AFL method only.

Program name	AFL First AV	AFL First EV	AFL + Method First converted EV	Convert time	Successful cases	Failed cases
CADET—00001	15"	19"	24"	9"	2	1
CROMU_00055	9"	29'51"	32"	23"	3	1
CROMU_00058	10"	/	45"	35"	1	2
CROMU_00064	7"	9'44"	53"	46"	2	0
CROMU_00078	54"	/	2'15"	1'21"	1	0
YAN01_00001	6'52"	6'38"	7'40"	48"	1	1
LEGIT_00003	3"	2'27"	21"	18"	1	0
TNETS_0002	7"	13"	39"	32"	1	1
NRFIN_00074	14"	/	1'24"	1'10"	1	0
NRFIN_00078	2"	2'22"	58"	56"	1	1
Equation_parser_overflow	9"	/	37"	28"	1	1
NRFIN_00059	34"	/	/	27'41"	0	6
KPRCA_00065	32"	36"	/	4'11"	0	3

variation, to improve the rate of exception conversion and reduce the dependence of exception conversions on initial test cases.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This project is funded by the National Science Foundation of China (61602502).

References

- [1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Pearson Education, Upper Saddle River, NJ, USA, 2007.
- [2] "American Fuzzy Lop," <http://lcamtuf.coredump.cx/afl/>.
- [3] "Peach Fuzzer," <https://www.peach.tech/>.
- [4] "beSTORM, Software SecurityTestingTool," <https://www.beyond-security.com/bestorm.html>.
- [5] "Crash dump analysis using BitBlaze," <https://www.utdallas.edu/~zxl111930/fall2011/lec6.pdf>.
- [6] D. Kim, X. Wang, S. Kim, A. Zeller, S. C. Cheung, and S. Park, "Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 430–447, 2011.
- [7] "Exploitable Crash Analyzer – MSEC Debugger Extensions," <http://msecdbg.codeplex.com/>.
- [8] "Apple Product Security. Announcing," CrashWrangler, 2009, <http://seclists.org/dailydave/2009/q3/11>.
- [9] "CERT Triage Tools," <https://www.cert.org/vulnerability-analysis/tools/triage.cfm?>.
- [10] S. Heelan and D. Kroening, *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*, MSc Computer Science, Univ, Oxford, London, U.K, 2009.
- [11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy, S and P 2012*, pp. 380–394, USA, May 2012.
- [12] E. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," *Usenix Sec*, vol. 8, no. 3, p. 25, 2011.
- [13] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing: SAGE has had a remarkable impact at Microsoft," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.
- [14] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Aeg: Automatic exploit generation," in *Proceedings of the Network and Distributed System Security Symposium*, pp. 74–84, 2011.
- [15] Team Shellphish, *Cyber Grand Shellphish*, Phrack article.
- [16] DARPA Cyber Grand Challenge, <https://cgc.darpa.mil/>.
- [17] Robot Hacking Game, <http://rhg.ichunqiu.com/>.
- [18] M. Prandini and M. Ramilli, "Return-oriented programming," *IEEE Security & Privacy*, vol. 10, no. 6, pp. 84–87, 2012.
- [19] "Writing JIT-Spray Shellcode for fun and profit, Alexey Sintsov," interesting way to bypass DEP and ASLR in browsers (not only) and Just-In-Time compilers was presented.
- [20] M. Wang, P. Su, Q. Li, L. Ying, Y. Yang, and D. Feng, "Automatic polymorphic exploit generation for software vulnerabilities," *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, vol. 127, pp. 216–233, 2013.
- [21] "REX," <https://github.com/shellphish/rex>.
- [22] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annu*, pp. 41–46, 2005.
- [23] Y. Shoshitaishvili, R. Wang, C. Salls et al., "SOK: (State of) the Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy, SP 2016*, pp. 138–157, USA, May 2016.
- [24] Antonio Bianchi, *Automatic Binary Exploitation and Patching using Mechanical [Shell] Phish*, HITCON Pacific, 2016.
- [25] Ruoyu Wang, "the owner of angr, private communication," the owner of angr, private communication.

- [26] “TCPDUMP 4.5.2,” <https://www.exploit-db.com/apps/973a25-13d0076e34aa9da7e15ed98e1b-tcpdump-4.5.1.tar.gz>.
- [27] “PEDA - Python Exploit Development Assistance for GDB,” <https://github.com/longld/peda>.
- [28] David Litchfield, *Variations in Exploit methods between Linux and Windows*, USA, 2003, Black Hat.
- [29] “cb-multios,” <http://github.com/trailofbits/cb-multios>.
- [30] “Cyber Grand Challenge,” The Analysis <https://www.youtube.com/watch?v=SYYZjTx92KU>.
- [31] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004*, pp. 298–307, usa, October 2004.
- [32] Result of CGC, <http://www.lungetech.com/cgc-corpus/cwe/cfe/>.
- [33] “The official website of CGC, in which page the difficulty level included,” <http://www.lungetech.com/cgc-corpus/cfe/rating/>.
- [34] “Result of RHG,” <http://www.csdn.net/article/a/2017-09-25/15932765>.