Construct exploit constraint in crash analysis by bypassing canary

Ning Huang, Shuguang Huang, Hui Huang, and Chao Chang

Citation: AIP Conference Proceedings **1864**, 020194 (2017);
View online: https://doi.org/10.1063/1.4993011
View Table of Contents: http://aip.scitation.org/toc/apc/1864/1
Published by the American Institute of Physics

Construct Exploit Constraint in Crash Analysis by Bypassing Canary

Ning Huang[1, a)], Shuguang Huang[2], Hui Huang[2] and Chao Chang[1]

[1]*Ph.D Team, Electronic Engineering Institute, Hefei 230037, China*
[2]*Department of Network Engineering, Electronic Engineering Institute, Hefei 230037, China*

[a)]Corresponding author: 809848161@qq.com

Abstract. Selective symbolic execution is a common program testing technology. Developed on the basis of it, some crash analysis systems are often used to test the fragility of the program by constructing exploit constraints, such as CRAX. From the study of crash analysis based on symbolic execution, this paper find that this technology cannot bypass the canary stack protection mechanisms. This paper makes the improvement uses the API hook in Linux. Experimental results show that the use of API hook can effectively solve the problem that crash analysis cannot bypass the canary protection.

Key words: selective symbolic execution; crash analysis; exploit constraint; canary mechanism; API hook.

INTRODUCTION

With the development of information technology, the security of software has been paid more and more attention. In recent years, a variety of software protection mechanisms emerge in an endless stream. At the same time, the exploits against these protection mechanisms are also evolving. Among the software vulnerabilities, the control-flow hijacking is undoubtedly one of the most harmful types of vulnerabilities. Traditional buffer overflows, use-after-free, and other common causes of vulnerabilities can lead to control-flow hijacking for programs. It is likely to lead to arbitrary code attacks after the program process is hijacked. The detection of software vulnerabilities, especially those that could lead to the control-flow hijacking vulnerability, is a work that must be done before a qualifying software is launched.

Symbolic execution is one of the most popular program testing techniques. Compared to static and dynamic symbolic execution, selective symbolic execution has its own advantages. The programmer can perform a symbolic execution by selecting a partial code of the binary program. In the meanwhile, the rest of the program still run with concrete values driven by the concrete execution. Selective symbolic execution alleviates the problem of path explosion to a certain extent, which improves the efficiency of search and detection to programs. However, in the actual implementation, the selective symbolic execution is faced with the contradiction between the accuracy of the variable symbolization and the automation.

On the basis of symbolic execution, some researchers have proposed program crash analysis by building exploit constraints for program. Typical program crash analysis systems are Automatic Exploit Generation (AEG) [1] and CRAX [2] [3]. This technique, by triggering the collapse of a binary program, constructs an exploit that can be run by the program to detect the availability of the vulnerability and provide guidance for the repair of the vulnerability. However, many crash analysis systems, including CRAX, is often run in the ideal environment which is lack of the buffer protection and some other protection mechanisms. In this case, the analysis of the binary program running under certain protection mechanisms will make it difficult to achieve the desired analytical effect.

In view of the above problems, this paper designs the dynamic link library which can be used in Linux. We call it init_env.so. By using the Linux API hook, this library can effectively solve the problem that crash analysis cannot bypass the canary mechanism.

Green Energy and Sustainable Development I
AIP Conf. Proc. 1864, 020194-1–020194-13; doi: 10.1063/1.4993011
Published by AIP Publishing. 978-0-7354-1542-3/$30.00

BACKGROUND

Selective Symbolic Execution

The symbolic execution was proposed in the 1970s, and its main idea was to use symbolic value instead of the concrete value to drive the program to execute. In the symbolic execution, the program variable is expressed as an expression consisting of symbolic values and constants, and the program output is represented as a function of the symbolic value. At present, the symbolic execution mainly has static symbolic execution, dynamic symbolic execution (which is also called concolic) and selective symbolic execution. Static symbolic execution is mainly for the source code of the program. In a static test, the program will not be executed. The test is using symbolic variable simulation through the analysis of the data flow and control flow of the program.

Concolic combines the characteristics of concrete execution and symbolic execution [4]. Compared to the static symbolic execution, the concolic uses the concrete value as the program input, and performs the symbolic execution while the program is executed. In the process of running the program, the concrete value is used to run, as well as the symbolic value to record the running state of the program, and ultimately build a program path constraints. When a program path is over, the constraint solver generates the appropriate test case and detects the other path by reversing the state of the branch path of the program.

Whether it is static symbolic execution, or concolic, the purpose is to search all the path of the program as much as possible, find out program defects, and improve the security of software. On the other hand, when we test the large-scale software by symbolic execution, due to the huge size of the software code, the detection process will inevitably encounter the path explosion problems, and reduce the efficiency of symbolic execution [5].

Selective symbolic execution improve the efficiency of symbolic execution detection based on the concolic. Figure 1 to S2E, for example, describes the operation of selective symbolic execution. S2E divide a program into different translation blocks when the program is tested [6]. The main process of S2E testing is following:

(1)The test staff will specify the program variable marked as a symbolic variable, and then use the concrete value as the input for concrete execution.

(2)When the process is encountering symbolic variables, then turn into the symbolic execution.

(3)When each translation block is finished, S2E will checks whether there is a symbolic value in the register or memory. If there is no symbolic value, then the next translation block will be started as concrete execution.

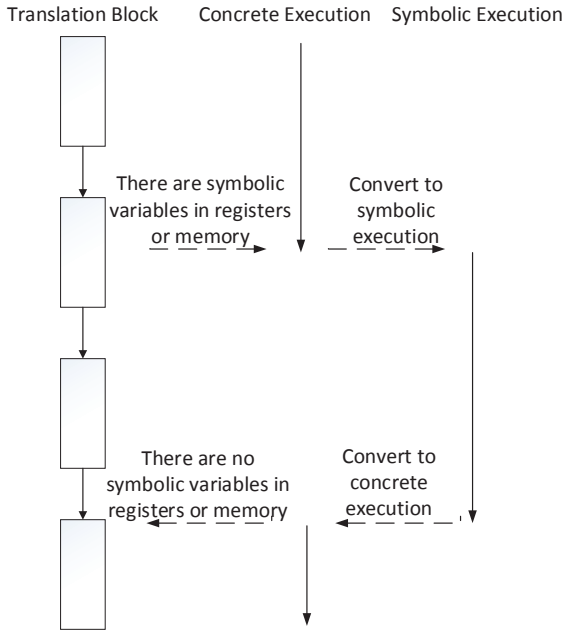

FIGURE 1. The conversion process between concrete execution and symbolic execution in S2E.

The Canary Mechanism in Linux

The modern operating systems have set various stack protection mechanisms against the traditional stack overflow which are exploited by covering the function address. Regardless of the cookie mechanism in Windows or canary mechanism in Linux, their principle are the same [7].

The commonly stack overflow exploitation is to make the length of variable in the stack frame longer than the length it was applied, so that it is sufficient to overwrite the function address at the bottom of the current stack frame and achieve the purpose of the process hijack. Figure 2 shows the two states of the canary in the stack frame.

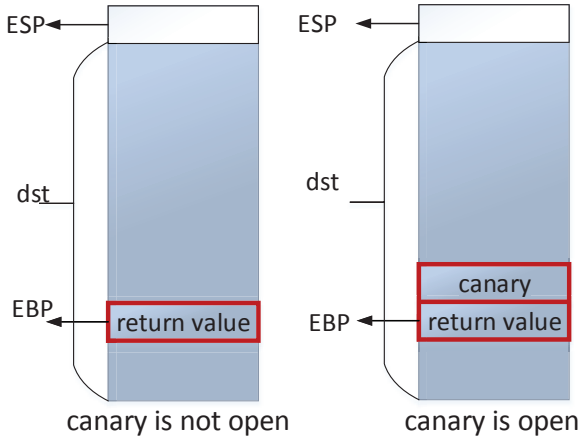

FIGURE 2. The two states of the canary in stack frame**.**

Before the application is running, the system will first call the _libc_start_main function in glibc for initialization. . One of the initialization jobs is to generate a canary random value with the _dl_setup_stack_chk_guard function and save the value in the memory space of the gs register offset 0x14 (the canary value in 64-bit Linux is stored in the fs register offset 0x28 in memory).

When a function is running, the system first checks the value of the canary above the return address and the value at [gs + 0x14]. If they are consistent, system will return the function. Otherwise, it considers that a stack overflow occurs and calls the _dl_stack_chk_fail function to handle the exception. The process is shown in Figure 3.

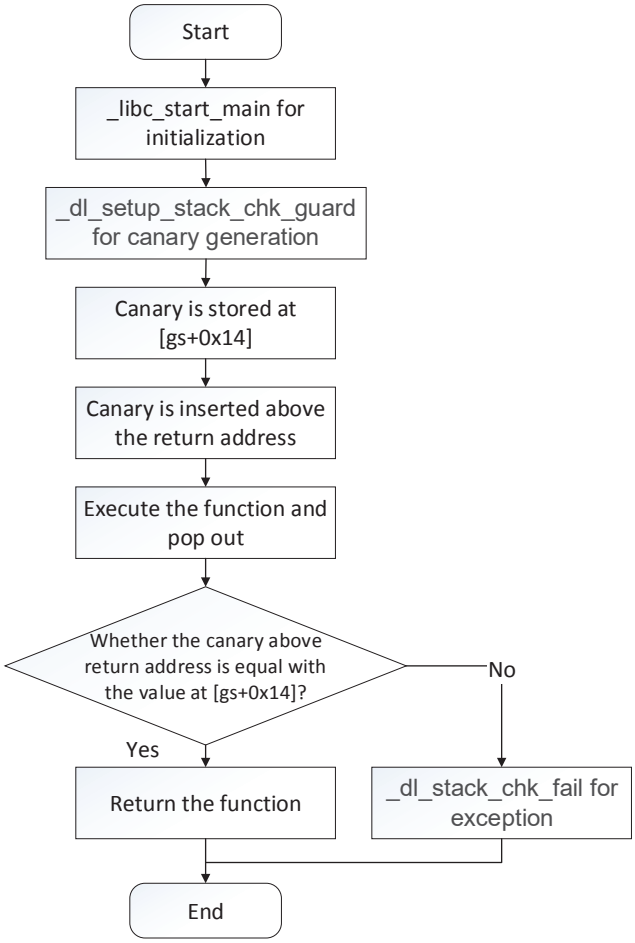

FIGURE 3. The canary mechanism workflow in 32-bit Linux.

Crash Analysis based on Selective Symbolic Execution

In the literature [2], Shih-Kun Huang proposed crash analysis based on selective symbolic execution. Based on this theory, the crash analysis tool CRAX was constructed. On the basis of the concrete execution, CRAX monitor the tested program running in QEMU [8] as a plugin of S2E. CRAX will convert to the symbolic execution mode after it detected the symbolic variable in the program. The symbol attribute of the program variable will be propagated in S2E, therefore, the EIP register status of the tested program is also marked as a symbolic value after the external data source is marked and the data trigger program crashes. If CRAX detects that the symbolized EIP value points to a memory address that cannot be executed, CRAX issues an on Eip Corrupt signal and converts to the crash analysis. Figure 4 shows the process of the transformation from symbolic execution to crash analysis.

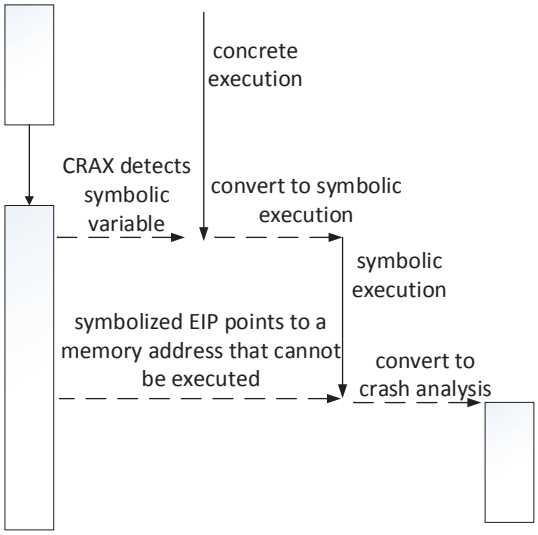

FIGURE 4. The process of the transformation from symbolic execution to crash analysis.

On the basis of the selective symbolic execution, CRAX implements the crash analysis of the control-flow hijacking vulnerability, constructs the feasible exploit, and provides a reference for the repair of the program. However, this technology must be used in an ideal system environment.

Assume that crash analysis is used to test the code in CODE 1. When running to the 10th line, the structure of the stack is shown in Figure 5.

CODE 1.

```
1  char  src[1024];
2  int main()
3  {
4      char  dst[16];
5      open(file);
6      read(file, src, 1024);
7      s2e_make_symbolic(src);  //symbolized the src
8      strcpy(dst, src);  //stack overflow
9      return 0;
10  }
```

If Code 1 is running under the protection of canary, the system will first check the value of the canary before the return address is read by the EIP register. If the check does not pass, the program will call the system exception handling function, and result in the failure of crash analysis.

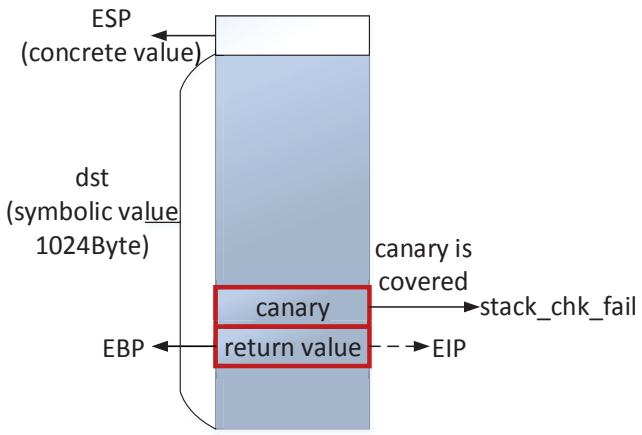

FIGURE 5. The stack frame of Code 1 under the protection of canary.

IMPLEMENTATION

In order to solve the problem that crash analysis cannot bypass the canary mechanism, we designed a new dynamic link library called init_env.so which can be used in Linux. This library uses the API hook in Linux. The crash analysis after loading the init_env.so library is shown in Figure 6.

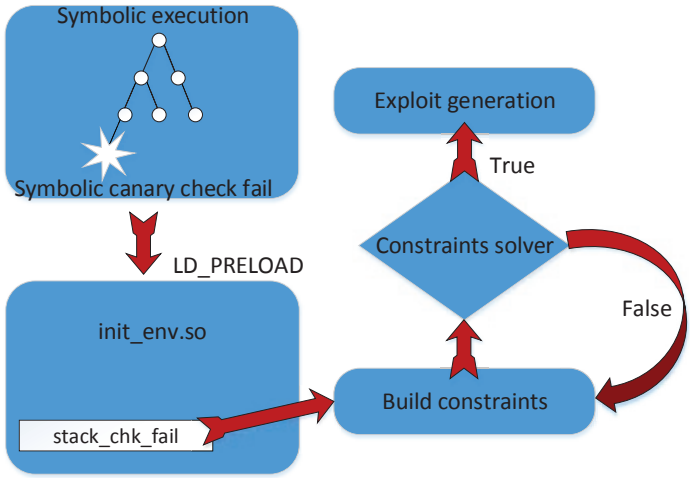

FIGURE 6. The crash analysis after loading the init_env.so library.

In order to make the tested program from the concrete execution convert to the symbolic execution, we need to determine that which variable should be symbolized. In the literature [10], Huang Hui proposed a symbolic execution technique based on the control-flow stain analysis. The technique uses stain analysis to mark external data sources and program variables that are contaminated by external data sources as symbolic values. Based on this, the tested program performs selective symbolic execution.

To monitor the external data input and stack overflow, init_env.so hooks some API functions, such as read, open and strcpy, access the pointer of these functions. Then, the program will preloads the custom init_env library preferentially by modifying the LD_PRELOAD environment variable.

Variable Symbolized based on Stain Analysis

The first work of crash analysis is to symbolize the variables. As everyone knows, the root of stack overflow, which is caused by control-flow hijacking vulnerability, is the unsafe operation of program on external inputs. Therefore, the init_env library insert the code to the program, to track the polluted variables and monitor the sensitive operations as the reference for variable symbolization.

On the basis of control-flow taint directed symbolic execution [10], we defines five properties for the variables in init_env.so to determine whether the variables should be symbolized. The attribute Q represents the result of the variable symbolization. The range of Q is 0 and 1, and when Q = 1, the variable needs to be symbolized; when Q = 0, it indicates that the variable does not need to be symbolized. The value of Q is determined by the other four attribute values of the variable: Attribute S indicates whether the variable is an external data source; attribute O indicates whether the variable is infected with external data; attribute I indicates whether the variable is infected by an internal source of contamination; attribute E indicates whether the variable has been purged. TABLE 1 lists the partial results of the attribute Q based on the other four attribute values.

$$Q =< S, O, I, E > \tag{1}$$

TABLE 1. Partial results of the attribute Q based on the other four attribute values.

Q	S	O	I	E
1	1	0	0	0
0	1	0	0	1
0	0	0	0	0
0	0	1	0	1
1	0	1	0	0

The definition of sensitive operations for stained data includes two types: assignment operations and sensitive function operations.

Assignment operation refers to the stained data spread through the assignment statement directly or indirectly. In the CODE 2, for example, the variable A is external data, and A is marked as the stain data according to the variable symbolization rule. After executing the second line of the program, the variable B is polluted by an assignment operation and becomes the stain data. Execute the third line of the program, B is assigned by a constant, so that B is purified, and no longer a stained data.

CODE 2.

```
1  read( A );
2  B = A;
3  B = 0;
```

Analysis of the spread of the stain caused by the operation of the sensitive function needs to be carried out by monitoring the process of the sensitive function call. Through the sensitive functions' hooks in init_env , we can effectively monitor the external data. The init_env marks these external data as sources of stain data and defines the rules for the propagation of stain data through sensitive functions. TABLE 2 lists some rules of the propagation of stain data through sensitive functions which are defined in this paper.

TABLE 2. Some rules of the propagation of stain data through sensitive function.

Function Name	Stain Source	Stain Target	Spread Rule
open(*a , b)	----	----	----
read(a , *b , c)	b	b	b
strcpy(*a , *b)	b	a	a←b

Path Selection

The second step of crash analysis is to traverse the program paths in symbolic execution. The path selection of crash analysis is based on the Depth-First Search (DFS).

The DFS for crash analysis is implemented by the search of the path state set of the symbolic path [9]. During the symbolic execution on the program, three sets of states are maintained: addStates, states, and removedStates. The three state sets are defined as follows:

addStates: The program state nodes that will be searched;

removedStates: The program state nodes that have been searched;

States: The program state node is searching.

The system traverses the program branch down from the parent node by comparing the three state sets. The search strategy is shown in Figure 7.

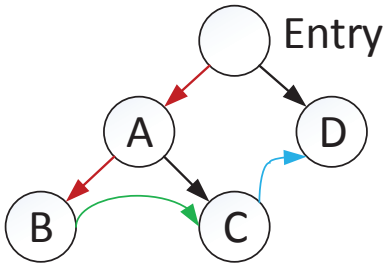

FIGURE 7. The search strategy of program state nodes.

The path search starts from the root node Entry. The Entry node contains two child nodes, means that it has two program paths under Entry. The system first marks the Entry state as fork, adds state A to the addStates set, and then executes the Entry node. At this point, the contents of the three state sets are as follows:

States = {Entry};

addStates = { A };

removedStates = { NULL };

After the execution of the entry, enter state A. State A also includes two child nodes B and C. The state set at this time is:

States = {Entry, A};

addStates = { B };

removedStates = { Entry };

According to the above path selection rules, KLEE implements the program path traversal based on DFS.

In the symbolic execution, if a loop statement is encountered, the loop will be converted to a conditional statement according to the program control-flow graph (CFG). Take the loop statement code shown in CODE 3 as an example.

CODE 3.

```
1    for( int i =0 ; i < 10 ; i++){
2      a[ i ] = i ;
3    }
4   return;
```

We can convert the loop module in Code 3 to the conditional branch statement as shown in Figure 8.

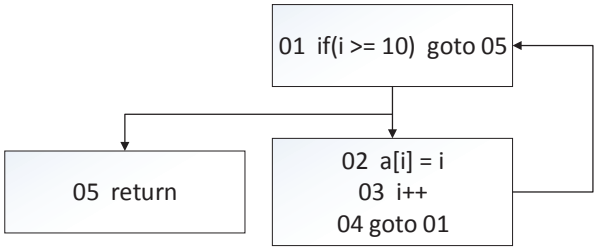

FIGURE 8. The loop statement in Code 3 to the control-flow of the conditional branch statement.

After the conversion from the loop statement to the conditional statement, the loop module will be insert to the path tree as a conditional branch state according to the DFS, and symbolic execute for this module.

Crash Analysis Bypass the Canary

After the two steps as described above, we can finally make the crash analysis bypass the canary and construct exploit constraints according to the analysis.

In the Linux system, LD_PRELOAD is an environment variable that specifies the priority of the dynamic link library loaded when the program is running. The dynamic link library loaded by LD_PRELOAD will be loaded first before the program runs. Thus, if the global symbols in the library specified by LD_PRELOAD are the same as the global symbols in the following libraries, the loaded libraries and the global symbols in them will be overwritten by the library specified by LD_PRELOAD. Because of this feature of LD_PRELOAD, it can affect the order of the link, allows users to customize the library which will be linked before running the program. Through the LD_PRELOAD, we can hook the API functions that the program will call.

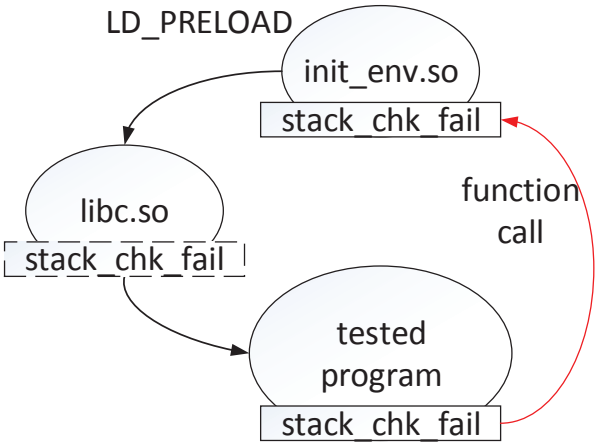

FIGURE 9. The API is hooked in init_env.so through LD_PRELOAD.

We redefine the exception handling function stack_chk_fail which is associated with stack overflow protection mechanism in Linux in the init_env. Preload the init_env library first by LD_PRELOAD, we can make the program execute the redefined exception handling function after triggering the stack overflow. The implementation process is shown in Figure 9.

After hook the stack_chk_fail exception handling function succesefully, we define the signature of stack_chk_fail function on the basis of S2E custom operation code mechanism. The signature allows the plug-in loaded in the host to identify the stack_chk_fail function that runs in the virtual machine, and trigger the condition that the cash analysis execute. The process of crash analysis is shown in ALGORITHM 1.

ALGORITHM 1. The algorithm of crash analysis.

```
Input: opcode (The signature of stack_chk_fail.)
       state (The current state of program.)
Output: result (The result of exploit generation.)
1  if( !opcode )
2      return;
3  symbolicArray[ ] = findSymbolicObj( state , 0x00000000 , 0xC0000000 ) ;
4  result = exploitGeneration( state , symbolicArray[ ] );
5  return result ;
```

When the host plug-in receives the signature of stack_chk_fail, it also gets the running state information of the program. The crash analysis algorithm uses the signature and state information as inputs to construct an exploit constraint to determine whether the program is eligible for the exploit.

Symbolic Memory Search

After determining the symbolic variables of the program by the stain analysis, we need to search the memory and find the symbolized memory for the exploit constraint. The symbolic memory search is shown in ALGORITHM 2.

ALGORITHM 2. The symbolic memory search algorithm**.**

```
Input:  state (The current state of program.)
        minAddr (The start address to be searched.)
        maxAddr (The end address to be searched.)
Output: symbolicArray [ ](The array of the symbolic memories.)
1  op = state->findObject;
2  for( i=minAddr ; i<=maxAddr ; i+=page_size )
3  {
4    for(objAddr=i; objAddr<=i+page_size; objAddr += obj_size )
5      {
6        if( is SymbolicObj( objAddr ) )
7          {
8            if( isPrevious SymboliObj( objAddr ) )
9                op.width++;
10           else  op.width = 1;
11         }else
12         {
13           if(isPrevious SymboliObj( objAddr ) )
14               symbolicArray[ ] = op;
15         }
16     }
17  }
18  return  symbolicArray[ ];
```

The symbolic memory search algorithm requires three inputs: the current state of the program, the start address of the memory to be searched, and the end address. According to the memory distribution of applications in 32-bit Linux, the starting address is generally set to 0x00000000, the end address is generally set to 0xC0000000. In this range, we will search the memory by page. The memory page size is defined as page_size in Algorithm2, this value must not be the memory page size as the Linux system defined, and we can custom it according to the needs. In each memory page, the symbolic memory is searched for in memory blocks of size obj_size. Similarly, the value of obj_size can also be customized according to the actual situation.

We need to make the following judgments to search a symbolic memory block:

(1) If the previous memory block is symbolic memory, the length of the symbol block is incremented by one;

(2) If the previous memory block is not symbolic memory, set the length of the symbolic block to 1.

When the searched memory block is not symbolic memory, the state of the previous memory block should be searched. If the previous block is symbolic memory, the information of the previous block is stored in the symbolicArray array.

Exploit Constraint Construction

After determining the address and length of the symbolized memory in the measured process, we constructs an exploit constraint on these symbolized blocks to determine whether the shellcode can be injected into the symbolized block. An exploit constraint consists of three parts: the shellcode constraint, the NOP constraint, and the EIP register constraint. This part will be achieved in the exploitGeneration in ALGORITHM 1. The specific process is shown in ALGORITHM 3.

ALGORITHM 3. The exploit constraint construction algorithm.

```
Input: state (The current state of program.)
Shellcode Addr (The start address of shellcode.)
Nomads (The start address of NOP.)
Output: exploit Constraints (The exploit constraints.)
1  for 0 ← shellcode.size
2      shellcodeConstraints.build( );
3  for 0 ← shellcodeAddr – nopAddr
4      nopConstraints.build( );
5  lowerBound = nopAddr;
6  upperBound = shellcodeAddr;
7  eipContraints.build ( lowerBound, upperBound );
8  exploitConstraints.build ( shellcodeConstraints ,
                              nopConstraints ,
                              eipConstraints );
9  s2e( )→solver( exploitConstraints );
10  if( exploitConstraints == true )
11      return exploitConstraints;
                12  else{
13      update( nopAddr );
                14      goto 1;
                15  }
```

The constructed exploit constraint will be submitted to the SMT constraint solver in conjunction with the program path constraint that arrives at the corresponding memory block to see if the program can reach this symbolized block.

Each symbolized block will be created by a symbolic expression to determine that each byte in the memory block can be replaced by the shellcode.

In the process of injecting shellcode into a symbolized block, to ensure that shellcode can be executed as much as possible, a null command (NOP) sequence is appended to the executable shellcode. In this process, the system creates a NOP constraint to determine the start and end positions of the NOP instruction in the symbolized block. The memory space occupied by the NOP instruction should be as large as possible. Therefore, before selecting the injected symbolized block, we will first sort all the symbolized memory blocks according to their size, and the shellcode will be preferentially injected into the largest memory block.

In order to ensure that the program can jump to symbolized memory blocks and execute shellcode, a new constraint must be established to guide the EIP register to the memory interval occupied by the NOP instruction.

By solving the exploit constraint by the constraint solver, it can detect whether the injected shellcode can be executed. If it can be successfully executed, the system returns a viable exploit constraint; if it is not feasible, the exploit constraint construction algorithm will be executed again after updating the start address of the NOP instruction.

EXPERIMENTAL RESULTS

Common Weakness Enumeration (CWE) is a set of international software defect description system. Compared with other vulnerability classification system, CWE analyse the vulnerability model and statistics its distribution, make this system more comprehensive and objective on the causes of vulnerabilities and practical impact.

In order to verify the validity of the crash analysis which is based on symbolic execution and can be used to bypass the canary, this paper uses the improved CRAX system to test the multiple stack overflow scenarios under CWE-121 system.

The name of the CWE-121 system is Stack Based Buffer Overflow. It lists a variety of stack overflow vulnerability scenarios. The crash analysis for these scenarios is undoubtedly instructive for the detection of stack overflow vulnerabilities in real environments.

TABLE 3. Different stack overflow scenarios in the CWE-121 system.

CWE ID	Bug Point	Shellcode Blocks	Reason
CWE-121	memcpy	2	Data length incorrect
CWE-129	if	1	Array overflow
CWE-131	memcpy	1	The target memory area does not exist
CWE-193	memmove	2	Data length incorrect
CWE-805	snprintf	2	Data length incorrect

TABLE 3 shows five different stack overflow scenarios in the CWE-121 system. Bug Point points out the location of the vulnerability. Reason indicates the cause of the vulnerability in the corresponding scenario. Shellcode Blocks shows the number of zones that shellcode is successfully injected into after crash analysis.

```
 99 6 [State 0] Timer starts
100 6 [State 0] 1 constraints in the state
101 6 [State 0] EIP is tainted by 0x1010101, original value is (
102 6 [State 0] Explore Time: 0 seconds
103 6 [State 0] 1 constraints in the state
104 7 [State 0] Found Symbolic Array at 0x804a060, width 992
105 7 [State 0] Found Symbolic Array at 0xbfc9b758, width 1
106 7 [State 0] Found Symbolic Array at 0xbfc9b78c, width 992
107 7 [State 0] Generating exploit on symbolic array 0x804a060
108 7 [State 0] ShellCode starts at 0x804a426, width 26
109 7 [State 0] NOP sled starts at 0x804a084, width 930
110 7 [State 0] Set EIP between 0x804a084 and 0x804a426
111 7 [State 0] Pruned 0 out of 1 constraints
112 8 [State 0] Write exploit to file exploit-804a060.bin
113 8 [State 0] Generating exploit on symbolic array 0xbfc9b78c
114 8 [State 0] ShellCode starts at 0xbfc9bb52, width 26
115 8 [State 0] NOP sled starts at 0xbfc9b7b0, width 930
116 8 [State 0] Set EIP between 0xbfc9b7b0 and 0xbfc9bb52
117 8 [State 0] Pruned 0 out of 1 constraints
118 8 [State 0] Write exploit to file exploit-bfc9b78c.bin
119 8 [State 0] Ended exploit generating
120 8 [State 0] Timer ends, took 2 seconds
121 All states were terminated
```

FIGURE 10. The results of the CWE-121 crash analysis.

Figure 10 shows the results of the CWE-121 crash analysis. In this paper, we construct a string with a length of 1024 bytes and a content of "00000000" as the input data of the measured program and the shellcode with a length of 26 bytes during the experiment. The input data successfully triggered the CWE-121's stack overflow. As can be seen from Figure 10, CWE-121 has three symbolized memory areas, where the starting address were 0x804A060 and 0xBFC9B78C. These two symbolic memory block is injected by the shellcode.

CONCLUSION

In this paper, we studied the principle of crash analysis based on symbolic execution and its implementation process. The study found that CRAX cannot bypass the canary mechanism in the process of crash analysis, and resulting in the exit problem. To solve this problem, we developed a new dynamic link library init_env.so which can be used in Linux.

We uses the API hook in Linux in init_env.so to help bypassing the canary mechanism in crash analysis. In order to verify the effectiveness of the above techniques, this paper tests some programs with stack overflow. The test results show that by loading the newly developed init_env.so library, the input of the program and the related stain data are accurately symbolized, and the canary mechanism is successfully bypassed. These experimental results achieve our desired goals.

REFERENCES

1. Avgerinos, Thanassis, et al. "AEG: Automatic Exploit Generation." Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, February 2011.
2. Huang, Shih Kun, et al. "CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations." IEEE Sixth International Conference on Software Security and Reliability, IEEE Computer Society, 2012:78-87.
3. Schwartz, Edward J, T. Avgerinos, and D. Brumley. "Q: Exploit Hardening Made Easy." USENIX Conference on Security USENIX Association, 2011:25-25.
4. S.Wu. “Software Vulnerability Analysis Technology.” 2014:333-335.
5. J.An. “Research on Key Technology of Dynamic Symbolic Execution.” Ph.D. thesis, Beijing University of Posts and Telecommunications, Beijing, China, 2014.
6. Chipounov, Vitaly, V. Kuznetsov, and G. Candea. "S2E: a platform for in-vivo multi-path analysis of software systems." ACM, 2011:265-278.
7. “Interpreting the stack security protection of the Linux security mechanism”, http://www.hackdig.com/03/hack-32670.htm.
8. Bellard, Fabrice. "QEMU, a fast and portable dynamic translator." Conference on USENIX Technical Conference USENIX Association, 2005:41-41.
9. Cadar, Cristian, D. Dunbar, and D. Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs." USENIX Conference on Operating Systems Design and Implementation USENIX Association, 2008:209-224.
10. H.Huang, et al. “A research on control-flow taint information directed symbolic execution.” Journal of University of Science and Technology of China. 2016:21-27.