Research on binary vulnerability mining technology based on control flow integrity detection

Bing Zhang
Information Engineering University

Abstract. Aiming at the problem of strong pertinence and poor universality of existing binary vulnerability detection methods, binary vulnerability automatic mining method based on control flow integrity detection is proposed. In view of the common binary vulnerabilities, such as stack overflow, heap overflow, function pointer tampering, structured exception handling attacks and so on, they can be unified and abstracted to destroy the integrity of control flow. This method first obtains all legitimate control flow transfers through static binary analysis, then uses symbol execution engine to explore all feasible paths, and injects constraints at control flow transfers. Finally, by solving constraints to determine whether there is a vulnerability that destroys the integrity of control flow. If there is, it automatically generates input that can trigger vulnerabilities.

1. Introduction
Under the background of current software security vulnerability and its corresponding security detection needs, this paper studies software vulnerability detection and analysis technology with high practical value. Taking binary file as the research object, this paper studies the relevant technologies of binary vulnerability analysis and detection. On the basis of the existing research results, adopt symbol execution technology to analyze binary programs in a fine-grained way. In view of current vulnerability discovery theory does not abstract model to certain kind of vulnerability, but uses different vulnerability discovery method for different vulnerabilities, the common binary vulnerabilities are abstracted as the destruction of control flow integrity. In this paper, the integrity of control flow is monitored to determine whether the program has vulnerability, and the vulnerability location and the input of triggering vulnerability can be located by using symbol execution technology. At the same time, the reliability of the system is tested. The overall process flow is shown in the Figure 1-1.
2. Overall Design

2.1 The Framework of Overall System

This chapter elaborates on the design and implementation of the automated loop mining system based on control flow integrity detection. The framework of overall system is shown in Figure 2-1. The main implementation process of the system is to load the binary executable file. Angr will first disassemble it and then translate it into the intermediate language VEX for static analysis. Through the analysis of the intermediate language, the indirect jump of block is obtained, and then the actual jump address of the block can also be achieved due to Angr’s powerful analysis capability of indirect jumps, which is a legal constraint. Then use dynamic symbolic execution to explore all possible program paths and inject constraints at the point where the control flow is transferred so that it is not equal to the original constraints. That is to say, let the programs don’t jump to the original address. Use the solver executed by the symbol to solve the problem and see if the constraint is executable.
2.2 Design Thought
This article abstracts the main common binary bugs, including stack overflow, heap overflow, function pointer tampering, structured exception handling attacks, etc. into the destruction of control flow integrity. Although the scene conditions triggered by the vulnerability are different, these vulnerabilities have typical features of violating control flow integrity. Based on the symbolic execution, this paper utilizes a powerful symbolic execution engine Angr to implement program analysis and symbolic execution.

3. Identification of Indirect Control Flow Transfer
The basic unit of Angr analysis is block, which creates a block object. In a compiler construction, a block is a linear sequence of code with no branches other than entries and exits. This restricted form makes the block very easy to analyze. The compiler usually breaks down the program into blocks as the first step in the analysis process. The block constitutes a vertex or node in the control flow graph. The code includes an entry point, where there is no code, which is the destination of any jump instruction in the program; it includes an exit which means that only the last instruction can cause the program to start executing code in different blocks. In these cases, each time the first instruction in the block is executed, the remaining instructions must be executed in order. The instructions to start a new block include: the entry point of the procedure and function, the target of the jump or branch, the "fall-through" instruction behind some conditional branches, the instruction after throw exception and the exception handler. The end instructions of blocks contain: unconditional and conditional branch; direct and indirect branch; returning to the calling procedure; the possibilities of throw exceptions; if the function cannot return normally, they can be located at the end of the block; returning to instruction itself.

3.1 Intermediate Language Analysis
Identifying the indirect control flow transfer module is to load the binary program into the Angr analysis platform, and then convert the binary program into the intermediate language VEX.

Angr tries to ensure that the intermediate language has complete semantic information, and finds the address of the block with indirect control flow transfer through fine-grained analysis of the intermediate language. Specifically, when we carefully analyze the intermediate language of each block in the process of program analysis using Angr, we find that the jump address inside some functions is not a specific fixed address written directly. Instead, the address of the next jump is directly represented by a symbol variable. The sample function is based on the command line input to become the parameter of the function callsites() and prints the corresponding function according to the corresponding numbers.
Next, we compile this function into a binary file and load it into the Angr analysis platform, call the Angr command to generate the CFG, and call the CFG method to get the address of each function, which can be used to locate the specific function inside the function. Here we print the function information of callsites, among which the Angr command used is shown in Figure 3-2:

```
#include<stdio.h>
void fun1(){
    printf("fun1\n");
}
void fun2(){
    printf("fun2\n");
}
void fun3(){
    printf("fun3\n");
}
void fun4(){
    printf("fun4\n");
}
#define FUN_NUM 4
void (*fun[FUN_NUM])(void) = {fun1, fun2, fun3, fun4};
void callsite(int x)
{
    if(x>0 && x<4) (*fun[x])();
}
void main(int argc, char** argv)
{
    int i = atoi(argv[1]);
    printf("%d\n", i);
    callsites();
}
```

Figure 3-1 Sample Function

The output results are shown below:

| Function         | Syscall: False | SP difference: 0 | Has return: True | Returning: True |
|------------------|----------------|------------------|------------------|-----------------|
| fun4[0x40060a]   |                |                  |                  |                 |
| Arguments:       | reg: [72], stack: [0L] |
| Blocks:          | [0x400630, 0x400621, 0x40060a, 0x40061b] |

The specific information of Angr analysis includes function address, system call information, whether it needs to return, register and stack information, and function block information. By finding
the function address, we can get the specific content of the intermediate language corresponding to the function through Angr, where the function address of callssites is 0X40060a. Through the information in the last line above, we have four basic block addresses with functions: [0x400630, 0x400621, 0x40060a, 0x40061b]. We can disassemble the intermediate language of the two blocks printed to see the specific information shown in Figure 3-3:

```
irsb = b.factory.block(0x40060a).vex irsb.pp()
Results:
IRSB {
  IMark(0x400619, 2, 0) ------
  t32 = 64 to 32 (0x0000000000000000)
  t33 = 64 to 32 (t19)
  t31 = 32(t33, t32)
  t30 = 1U to 64 (t31)
  t26 = t30
  t34 = 64 to 1 (t26)
  t21 = t34
  if (t21) { PUT(rip) = 0x400630L; Ijk_Boring }
  NEXT: PUT(rip) = 0x000000000040061b; Ijk_Boring
}
```

```
irsb = b.factory.block(0x400621).vex irsb.pp()
Results:
IRSB {
  IMark(0x40062e, 2, 0) ------
  t19 = GET:I64 (rsp)
  t18 = Sub64 (t19, 0x0000000000000008)
  PUT(rsp) = t18
  STle(t18) = 0x0000000000400630
  t20 = Sub64 (t18, 0x0000000000000080)
  if (t20) { AbiHint (0xt20, 128, t15) }
  NEXT: PUT(rip) = t15; Ijk_Call
}
```

Figure 3-3 Specific Information of the VEX Intermediate Language in the Block

### 3.2 Address Acquisition

From the information in the figure we can see that the last statement of the first block is PUT(rip)=0X40061b. After conversion to intermediate language, all control flow transfers are translated into PUT covering jmp, jnz, call, ret, etc for better analysis. It can be seen that all PUT(rip) of this block are constants, and if the IF is completed after the block ends, it will jump directly to 0X400630 to continue execution. This is another block of callssites with boring type. If not, it will jump to 0x40061b. When it turns to the second block, the last statement the call of callssites, is also translated into PUT (rip), but the assignment is a variable t15, and is no longer a constant, so this is an indirect jump. If the attacker can control t3, he can tamper with the function return address. Therefore, the direct jump is PUT (rip) = CONST, and the indirect jump / call is PUT (rip) = VAR. The idea of this paper is to use CFG generated by Angr to print out the information of each node, which includes the address of each block of the program. Then the printed node information is input into the text file, and then we read it out and have regular matching of the text. The advantage of regular matching is that the information can be quickly and accurately extracted, and all the address information therein is extracted and stored in an array. The algorithm code is shown in Figure 3-4.
4. Identification of All Legitimate Targets for Indirect Control Flow Transfer

4.1 Processing of Side Information of Control Flow Graph

The main function of CFGAccurate is to generate static control flow and recovery program. Starting with the entry point of the program (or any user-defined point), the process is roughly as follows:

1. Converting the basic block language to the intermediate language VEX and collect all information about the exit (jump, call, return, or continue to jump to the next block);
2. For each exit, if it is a constant address, an edge is added to the corresponding CFG, and the target block is added to the corresponding set for analysis;
3. With regard to a function call, the block of the target program is regarded as the start of a new function. If the target function is known to return, the block after the function call is also analyzed.
4. In the case of a return, the current function is marked as return and the corresponding edges in the callgraph and CFG are updated.
5. For all indirect jumps (the jump address is not a constant), implement an indirect jump resolution.

The generation of CFG has a variety of ways, that is, you can choose any point as the starting point. The first step is to analyze the main entry point of the binary file. For binary files with symbols (for example, non-stripping ELF and PE binaries), all function symbols will be used as possible starting points. For binaries without symbols, such as stripped binaries, or binary files loaded by using the blob loader, CFG scans the binaries for a set of function prologs for the binary's architecture. Finally, the entire code portion of the binary is scanned by default, only by looking up the executable content rather than the prologue or symbol. This inferred control flow edge is called "FakeRet". If it is found that this is not true when analyzing, it will update CFG, delete "FakeRet", and update the callgraph and function block accordingly. Therefore, the CFG will be overwritten twice. During this process, the blocks in each function can be directly restored and passed regardless of whether the function returns. The CFG is shown in Figure 4-1.

```python
1. b = angr.Project("Filename", load_options=auto_load_libs=False) # Loading binary files
2. cfg = b.analyses.CFGAccurate(keep_state=False) # Generating CFG
3. f = open('CFGnodes.txt', 'w')
4. for node in cfg.nodes(): # Getting specific information for each node and print it to a text file
5.    line = f.read() # Wrapping with a comma
6.    s1 = []
7.    lines = line.split(',') # Wrap with a comma
8.    s = re.findall('0x\w{5,20}', node) # Regular match for each line because the block address starts with 0x
9.    for node in lines:
10.   s1.append(s[0]) # The address is stored in an array
11.   blocks = s1
12.   irsb = b.factory.block(block).vex # Getting specific information of intermediate language for each block
13.   f = open('irsb.txt', 'w')
14.   for node in lines:
15.       s = re.findall(r'(t.*', node) # To see whether the last jump address of regular match is a symbol or not

Figure 3-4 Algorithm Code of Indirect Jump Address Acquisition
4.2 Address Information Acquisition

In the case of obtaining an accurate control flow graph, an accurate method should be designed to get right address. Through analysis, we can know that the obtained side information is accurate enough under the existing laboratory conditions, so we need to use a reliable extraction method. Here, we carefully select the regular expression to extract the address after research. Regular expressions are very flexible, logical, and functional, and can quickly achieve complex string control in a very simple way. The binary programs designed and tested in this paper have many functions and the logical structure is not clear, so the obtained control flow graph will be very complicated, and the corresponding side information is very cumbersome. Hence, the use of regular expressions can well meet the needs of this paper. The flow of the algorithm is shown in Figure 4-2.
Specific Information of Edge

5. Path Exploration and Injection of Constraint Solving to Determine Whether There is a Vulnerability

5.1 File Processing of Angr

Angr is capable of execution scheduling and analysis simulations on loaded binary programs. It uses the CLE loader to load the entire collection of binaries, load and map to a single storage space. Each binary is loaded by a loader backend that can handle its file type (a subclass of cle.Backend). For example, cle.ELF is used to load ELF binary files. There are also objects in memory that do not correspond to any loaded binaries.

CLE currently has a backend for statically loading ELF, PE, CGC, Mach-O, and ELF core dump files, as well as a backend for loading binary files using IDA and loading files into a flat address space. Data types such as floating point values and Boolean values can be represented by AST, and any Boolean value can be used as an assertion of the valid value. The method is to add it as a state constraint and then query the valid value of the symbol variable by asking for the symbol variable to be evaluated. By adding these constraints to the state, the constraint solver treats them as any assertions that must satisfy their return values.

State is a very important component of Angr. State keeps various state information during the execution of Angr’s dynamic symbols. These information are indispensable parts for symbolic execution. Figure 5-1 shows the whole process of State:

![Figure 5-1 state](image)

Use state to describe the specific state of an executed address at the same time use SimulationManagers to manage how a program goes from one state to another. It is the most important interface for analog control programs in Angr. It can control a set of states for symbolic execution at the same time, and can also apply exploration strategies to explore the state space of the program. SimulationManagers handles multiple states in a clever way, and States are organized as "stashes," where forward steps, filters, merges, and moves can be done. This allows similar stashes that belong to different states to be executed at different rates and then merged. Most operations of stash default to the active state.
5.2 Implementation Ideas and Processes

The method in this paper is to perform symbolic execution on each block with indirect jumps, use the explore method to search, and use Angr's powerful and highly flexible symbol processing engine to get the specific state information of the current block which includes various registers, memory, and values of the constraint expression for a symbol variable mentioned earlier. Then, using the step method on the state information of the obtained indirect jump block, the currently executed program is taken one step forward, and the program executes all the stash information. The flowchart of the algorithm is shown in Figure 5-2:

1. When setting the symbol variable, according to the specific situation of the binary program, the number and size of the symbol variable are set in advance;
2. Remove the Angr default LAZYSOLVES option, because this option allows Angr to default to the analysis program unless it is absolutely necessary, otherwise it will not check whether the current state meets the operational requirements. Once this option is removed, Angr will check if it is satisfied every time there is a state update, so the accuracy will be greatly improved;
3. Keep debugging according to different programs. As long as there is a path to found, it will be extracted when collecting state information, so that the obtained path information will not be missed.

For how to add constraints, the method adopted in this paper is to make the extracted constraint information not equal to the original jump address, and then put back the path state to let the solver solve the program's running state and to see if the path state can be satisfied or not. As long as it is satisfied, it shows that the program can run not according to the original control flow, and can jump to other places to continue running, which illustrates the program is vulnerable. For the accuracy and reliability of the solver, because the solver used by Angr is well recognized internationally, the accuracy of all the experimental results has been verified by the related paper.

The running result of the program is shown in Figure 5-3.
6. Chapter summary

This chapter details the details of binary vulnerability mining technology implementation based on control flow integrity detection. Firstly, the overall framework structure and design ideas are introduced. Then, how to realize the identification of indirect control flow transfer and then identify all the legitimate targets of indirect control flow transfer, after doing this, use the symbol execution engine to perform the path of binary program simulation execution. Exploring, explaining the various technologies used and how to implement them. After obtaining the desired program state, the code written in this article forcibly changes the constraint, and then uses the constraint solver to determine whether the modified constraint can satisfy the program running condition. If the condition is satisfied, the symbol variable value used therein is solved. After solving it, inject the result into the source program to observe the error message. Each implementation module has been sorted and displayed.

References

[1] J. Song, J. Alves-Foss. The DARPA Cyber Grand Challenge: A Competitor’s Perspective, Part [J]. IEEE Security & Privacy, 2016, 14(1):76–81
[2] 0day[EB/OL]. https://baike.baidu.com/item/0DAY, 2019
[3] L. Qi, J. Wen, H. Huang, et al. The research on the fuzzing[M]. 2013, 85–91
[4] P. Godefroid, N. Klarlund, K. Sen. DART: directed automated random testing[M]. 2005, 213–223
[5] J. C. King. Symbolic execution and program testing[J]. Communications of the ACM, 1976,19(7):385–394
[6] P. R’ummer, T. Wahl. An SMT-LIB theory of binary floating-point arithmetic[M]. 2010, 151
[7] D. Liew, D. Schemmel, C. Cadar, et al. Floating-point symbolic execution: A case study in -version programming[M]. 2017, 601–612
[8] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[M]. 2008, 209–224
[9] L. De Moura, N. Bjørner. Z3: An efficient SMT solver[M]. 2008, 337–340