Hindawi
Security and Communication Networks
Volume 2021, Article ID 5581274, 14 pages
https://doi.org/10.1155/2021/5581274

Research Article

A Detection Approach for Vulnerability Exploiter Based on the Features of the Exploiter

Jinchang Hu,[1,2] **Jinfu Chen**,[1] **Sher Ali**,[1] **Bo Liu**,[1] **Jingyi Chen**,[1] **Chi Zhang**,[1] **and Jian Yang**[1]

[1]*School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China*
[2]*Command and Control Engineering College, Army Engineering University of PLA, Nanjing, China*

Correspondence should be addressed to Jinfu Chen; jinfuchen@ujs.edu.cn

Received 24 February 2021; Revised 17 April 2021; Accepted 6 May 2021; Published 22 May 2021

Academic Editor: Ke Gu

With the wide application of software system, software vulnerability has become a major risk in computer security. The on-time detection and proper repair for possible software vulnerabilities are of great importance in maintaining system security and decreasing system crashes. The Control Flow Integrity (CFI) can be used to detect the exploit by some researchers. In this paper, we propose an improved Control Flow Graph with Jump (JCFG) based on CFI and develop a novel Vulnerability Exploit Detection Method based on JCFG (JCFG-VEDM). The detection method of the exploit program is realized based on the analysis results of the exploit program. Then the JCFG is addressed through combining the features of the exploit program and the jump instruction. Finally, we implement JCFG-VEDM and conduct the experiments to verify the effectiveness of the proposed method. The experimental results show that the proposed detection method (JCFG-VEDM) is feasible and effective.

1. Introduction

With the development of society, the computer network has taken roots in every direction of the society as an essential part of modern life. However, there is no effective detection method for existing malicious programs in the network. People enjoy the convenience brought by computer technology but do not have an effective method to prevent the exploit programs [1–3]. At present, researchers have made some achievements in this field, but in the face of the endless stream hijacking attacks, the current exploit detection methods are still lacking pertinence [4–6]. Therefore, in view of the current vulnerability exploit attacks and their variants, we propose an improved Control Flow Graph (CFG) incorporating the features of the exploit programs. It is of great significance to the detection of exploit programs, beneficial to researches on the detection method of exploit programs. We also propose a Vulnerability Exploit Detection Method based on CFG with Jump (JCFG) (JCFG-VEDM).

The main contributions of this paper are as follows:

(1) The feature definition of the exploit for the abnormal jump is proposed based on empirical analysis.

(2) Under the premise of understanding the attack principles of existing exploits, we analyze the features of exploits, integrate the feature information into the CFG, and add pointer-related concepts. In addition, we propose the JCFG by combining the Control Flow Integrity (CFI) detection method.

(3) Based on the research of vulnerability features, this paper also proposes the JCFG-VEDM and focuses on the JCFG based on the features of the vulnerability exploit generation and detection algorithm of exploit programs. The experiments have shown that JCFG-VEDM has good feasibility and effectiveness in detecting abnormal jumps of exploit programs.

The rest of the paper is organized as follows. In Section 2, we describe the related work. In Section 3, we introduce the analysis method of the exploit based on program features and Section 4 proposes the vulnerability exploit detection method based on the features of vulnerability exploit. The

experimental analysis is reported in Section 5. Conclusions are presented in Section 6.

2. Related Work

The current exploit detection methods mainly use the control flow of a program to detect and protect the program, including CFI and taint analysis.

The original intention of CFI is to eliminate control flow hijacking attacks. In 2005, CCS (ACM Conference on Computer and Communications Security) published a paper called "Control Flow Integrity (CFI)" proposing the concept of CFI [7, 8].

CFI detection is divided into fine-grained CFI and coarse-grained CFI. The fine-grained CFI is proposed primarily, obtaining the corresponding CFG through static analysis of the program, calculating the destination address that the jump instruction may reach, and assigning an address ID to each address. Whenever the program jumps, it is logically checked to check whether the jump is a legal address [9, 10]. For example, XFI [11] modularized the program by combining memory access mechanisms to protect and monitor the program while it is running. However, due to the fine-grained CFI imposes too much overhead on the system, it is difficult to implement. The CFI program proposes a simplified version of the solution, which is called coarse-grained CFI [12, 13]. Coarse-grained CFI neither needs to obtain the CFG of the program nor needs to assign a corresponding address ID to each address where the program jumps. It only needs to perform a static analysis on the program and calculate the legal transfer addresses through the corresponding rules [14–16]. For example, CCFIR (Compact Control Flow Integrity and Randomization) [17] collects all legal transfer addresses together, and the program can only jump between these legal addresses. CFIMon [18] uses static analysis of the program to obtain the legal access addresses and then uses the score tracking storage mechanism (LBR, etc.) in the processor to detect the program and analyze the CFI of the program in real time. For all that, these coarse-grained CFI detection methods can reduce system overhead, the jump instruction and the return instruction cannot be one-to-one correspondence due to the fact that each jump address is not assigned a corresponding address ID, which leads to a call instruction that can enter any function. In the beginning, a vulnerability is created.

At present, among the various techniques for program analysis, researchers in the field of program analysis prefer stain propagation analysis, which is combined with the analysis technology to analyze the program for a more accurate program analysis report. Stain analysis is divided into static analysis and dynamic analysis [19]. Static Taint Analysis is to analyze the program statically without running the program to detect whether the data can be transmitted from the source of the taint to the spot [20–22], whereas Dynamic Taint Analysis is to detect whether the data can be transmitted from the source to the aggregation point while the program is running [23, 24].

In recent years, the academic circles mainly use CFI, stain analysis, and other detection schemes to detect the exploit [25, 26]. These schemes detect the abnormal control flow during running the program and have achieved certain results in practical application. However, the integrity detection of control flow needs to deal with the program at code level. Taint analysis mainly monitors the dynamic execution process of the program and sends out an alarm when the tainted data are used abnormally. The detection results have a certain degree of error because of the method's own limitations. These methods need instrumentation in the program and have excessive system overhead. Therefore, in view of the abnormal jump in the exploit, it is of great significance to deeply analyze the features of the exploit and provide the definition and unified formal description of the features of the exploit, which can further promote the research on the detection of the exploit, and the approach that we proposed does not need instrumentation, making security researchers more convenient to detect the exploits.

3. Analysis Method of Exploit Based on Program Features

The program features of the exploit rely mainly on the result of abstracting the program features of the exploit identified by the program feature. So far, the research on the program features of the exploit is not mature enough. The feature definition and formal description of the exploit are beneficial to the research of the exploit detection method.

This section first analyzes the features of the exploit and then the JCFG is proposed. Finally, the exploit is formalized through JCFG in this section.

3.1. Definition of Exploit Features

3.1.1. Lexical, Grammatical, and Semantic Features. There are diversified ways of exploiting exploits, with the main modality referred to the use of some specific instructions to achieve the attack from the perspective of assembly code. Therefore, we analyze the exploit from three aspects: lexicon, grammar, and semantics. For example, for each transfer instruction, when the transferred address does not exist in the legal transfer address, the node of the transfer instruction constitutes a dangerous node of exploit. An exploit example with C source code is shown in Figure 1.

This program compares the string in the file with PASSWORD, verifies whether it is consistent, and outputs the result. In this program, there is a vulnerability in the buff array in the verification function. By overwriting its return address, the program can jump to the starting address of the shellcode to perform related operations. Its corresponding assembly code flowchart is shown in Figure 2.

In Figure 2, the exploit can direct the control flow of the program to the address of the shellcode by overwriting the return address of the verity function. This is the dangerous node of exploit. Figure 2 mainly shows that the structure block of the verity function is mainly shown without some other system function calls. The call, jmp, jz, jnz, retn, and so on are instructions in the assembly code which constitute the grammatical features of the exploit. Through summarizing

```
1. #include <stdio.h>
2. #include <windows.h>
3. int verity (char *password)
4. {
5.       int result;
6.       char buff [8];
7.       result = strcmp (password, password);
8.       strcpy (buff, password);
9.     return result;
10. }
11. void main ()
12. {
13.       int flag = 0;
14.       char password [1024];
15.       FILE * fp;
16.       if (! (fp = fopen (“password.txt”, “rw+”)))
17.       {
18.             exit (0);
19.       }
20.       fscanf (fp, “%s”, password);
21.       flag = verity (password);
22.       fclose (fp);
23.       if (flag)
24.       {
25.             printf (“false.”);
26.       }
27.       else
28.       {
29.             printf (“true.”);
30.       }
31. }
```

Figure 1: An exploit example with C source code.

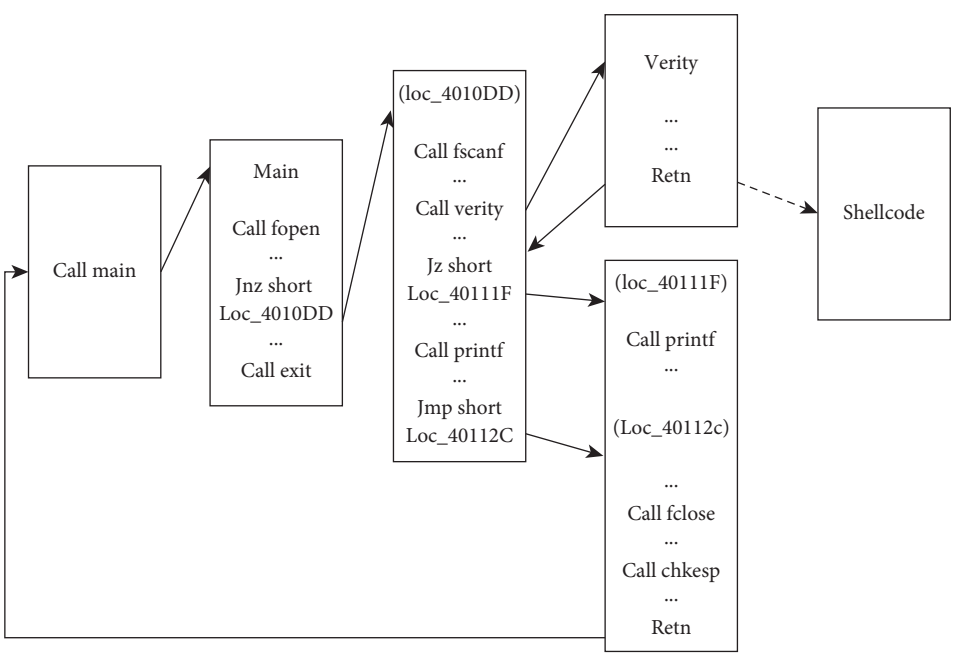

Figure 2: The flowchart of the vulnerable assembly code.

the grammatical features of these instructions, the following points are obtained: (1) function call instruction Call; (2) address transfer instruction JXX; (3) return instruction Return. In this paper, we refer to the grammatical features of these exploits as the dangerous element of exploit represented by σ. The dangerous node of exploit must contain the dangerous element of exploit. Therefore, we mainly analyze the address of the dangerous element of an exploit in the program to determine whether it is dangerous.

The previously mentioned dangerous element of exploit is shown in Figure 2. By analyzing the location of the dangerous element of exploit, it can be judged whether it belongs to the dangerous node of exploit, when the block program pointed by the dotted line of the program is executed in Figure 2. In this paper, a collective symbol D is established to contain all the dangerous nodes of exploit in the program. Afterwards, we will further analyze the dangerous node of exploit to determine whether they belong to the exploit nodes. In addition, the set of exploit nodes is denoted as V.

3.1.2. Definition Description. Albeit many ways to exploit, generalizing them from the perspectives of grammar, morphology, and semantics can always find a certain similarity. In this paper, the program features of the exploit are denoted as μ, and the constraint of the exploit is denoted as C. The program features of the exploit in this paper are described in Definition 1.

Definition 1. Program features of the exploit μ: $\mu(\text{Vul}) = \{D, C\}$. Vul represents the type of vulnerability exploited by the exploit. Prog represents the program containing instructions for the dangerous element of exploit σ, that is, the dangerous node of exploit. D is the collection of these dangerous nodes of exploit, such that, $D = \{d_1, d_2, \ldots, d_n\}$. C represents the relevant vulnerability exploit constraints that the features of the exploit program need to meet, such that $C = c_1||c_2||\cdots||c_j||\cdots||c_n$. For an exploit, the basic constraint Bc_j and the additional constraint Tc_j of the exploit need to be satisfied, meaning $c_j = Bc_j \wedge Tc_j$.

In the program feature μ of the exploit, D describes the performance and syntax features of the exploit, and C describes the semantic feature of the exploit. And, for the program feature of the exploit, it also has the following properties.

Property 1. The number of nodes is limited for a program. Accordingly, the number of dangerous nodes of exploit is also limited.

Property 2. For the dangerous node of exploit, the node must contain the dangerous element of exploit.

3.2. Formalization of Exploit Features. The feature form of the exploit refers to the formal expression of the feature of the exploit, which provides a strong foundation for describing the exploit in more detail and facilitates the research on the detection of the exploit. The main research object of this section is the exploit of abnormal jump.

3.2.1. Control Flow Graph Based on Jump (JCFG). At present, most of the detection methods for exploit used by researchers are to design corresponding exploit detection algorithms through the CFG of the program for detecting the exploit. The main detection method for exploit is CFI, which is divided into fine-grained CFI and coarse-grained CFI. For fine-grained CFI, it allocates a unique ID for each instruction jump and adds the detection function to the program. The system overhead is exceedingly large, and the efficiency cannot meet real needs. Therefore, the researchers proposed the coarse-grained CFI which does not need to assign a unique ID to each jump instruction but needs to detect whether the address of each jump is in the legal address set. However, the coarse-grained CFI has a certain impact on the accuracy of detection. Therefore, based on the strengths and weaknesses of the above two exploit detection methods, this paper proposes a new control flow graph based on the CFG by combining the features of the exploit, called the Control Flow Graph-based Jump (JCFG). In JCFG, only dangerous nodes of exploit are included. For each dangerous node of exploit, the node in JCFG mainly contains the following attributes: (1) instruction type, including jmp, call, jz, jnz, and retn; (2) the name of the called function; (3) jump address. These attributes are recorded as the feature attributes of the node.

Definition 2. Control Flow Graph based on Jump (JCFG): JCFG = $(D, E, R, \text{Begin}, \text{End})$. D represents the set of dangerous nodes of exploit contained in JCFG. For the dangerous node of exploit d in the set, $d = (\text{id}, \text{attr}, \text{next}_{\text{id}})$, id represents the number of the node in JCFG, attr represents the feature attribute of the node, and next_id represents the node that the current node points to. For next_id, there may be a forked path, so the first node pointed to is marked as $*$first, and the second node pointed to is marked as $*$second. For node attributes, address represents the current address of the instruction, attrName represents the name of the instruction, funcName represents the name of the function, and jAdress represents the jump destination address. E represents the combination of edges, used to express the direction relationship between nodes. R represents the set of return addresses.

$$\text{attr} = (\text{address}, \text{attrName}, \text{funcName}, \text{jAdress}). \quad (1)$$

For each call instruction, the address after instruction is called is added to R. Begin is the entry node, and End is the end node of JCFG.

3.2.2. Related Definitions

Definition 3. Call instruction: Call represents call instruction.

Definition 4. Jump instruction: JXX represents the jump instruction, which includes the conditional jump instruction

JCC (where CC represents the character sequence of the test condition type, including jz and jnz) and the unconditional jump instruction jmp.

Definition 5. Return instruction: Return represents return instruction, including RETN and RETF return instructions. RETN represents return from the subroutine transferred in the segment, and RETF represents return from the subroutine transferred in between segments.

Definition 6. Return address set: each time a function call instruction is executed, the address following the call instruction is stored in the return address set R.

Definition 7. Node judgement: for the program, there may be remerging the two execution paths after jnz, so the judgement is made to avoid duplication. The judgement function is recorded as isSame (jAdress).

Definition 8. JCFG node pointer: $^*p$ represents the current JCFG node pointer pointing to the Begin node by default. After the main function is executed, $^*p$ points to the first dangerous node of exploit under the Begin node.

3.2.3. Example Analysis. Figure 3 shows the code segment of a simple program. This program is simplified based on the code segment of the exploit given in Figure 1. Its specific assembly code flow graph is shown in Figure 4, and the JCFG formed is shown in Figure 5.

First, the Begin node is obtained according to the main function, and the next address of the call instruction in the return address set R is stored. Then the conditional jump instruction in the JXX instruction is matched, forming the d_1 node through it. Each time the jump instruction JXX is matched, the same node determination function is called, and the same node is checked first. If it exists, the node pointing to it is directly pointed to the existing node. If it does not exist, then it is examined whether it exists if the node with the same jump destination address exists, and the next node id of the jump destination address path is added to the next_id of the node. The d_1 node has two successor nodes. The process of d_1 node is as follows: (1) select a successor path of d_1; (2) read the next call instruction; (3) put its next address into the return address set R; (4) get the d_2 node, read two call instructions and in turn, get d_3 and d_4 nodes, and put the next address of their call instruction into the return address set. The next matched instruction is the conditional jump instruction in the JXX instruction, through which the d_5 node is formed. The d_5 node has two successor nodes. The process of d_5 node is similar to d_1 node, generating the d_6 node. The subsequent matched instruction is the unconditional jump instruction in the two JXX instructions. The d_7 node and the d_8 node are formed, respectively, and then the matched instruction is found to exist through the same node judgement function, so the d_8 node points to the d_1 node. Then, it returns to another instruction path of the previous d_5 node. Followed by a call instruction and a JXX instruction, the corresponding d_9 and d_{10} nodes

```
1.  #include <stdio.h>
2.  #include <windows.h>
3.  int verity (char* password)
4.  {
5.     int result = strcmp (password, PASSWORD);
6.     return result;
7.  }
8.  void main ()
9.  {
10.    int flag = 0;
11.    char password [1024];
12.    while (1)
13.    {
14.       printf ("input password:  ");
15.       scanf ("%s", password);
16.       flag = verity (password);
17.       if (flag)
18.       {
19.             printf ("false.");
20.       }
21.       else
22.       {
23.             printf ("true.");
24.             break;
25.       }
26.    }
27. }
```

Figure 3: Simple program code segment.

are obtained, the following instruction is a Call instruction to get the corresponding d_{11} node, followed by a return instruction Return, and there is only one return address in the return address set R. Therefore, the node is determined as the End node. Backing up again, another instruction path can access the d_1 node, through the same node determination function, d_{11} can be added to the next_id set of d_1. This is the end and the JCFG graph is generated.

Here is just a brief description of the generation of JCFG. The specific JCFG generation algorithm will be introduced in detail in the next section.

3.3. JCFG Generation Method Based on the Characteristics of the Exploit

3.3.1. Control Flow Graph Generation Method by IDA. For the exploit to be detected, with the static analysis of the exploit, the exploit is imported into IDA, getting its assembly code and reading its instructions. The efficiency of generating JCFG directly by extracting instructions from the assembly code of the program to be tested is extremely low. Therefore, the process of generating the JCFG is as follows: (1) use IDA scripts to generate corresponding CFG; (2) use the CFG that generated by IDA to filter out some unimportant instructions and retain the required Call, JXX, Return, and other key Command; (3) process the generated CFG to obtain the required JCFG. The node information structure of the CFG is shown in Figure 6.

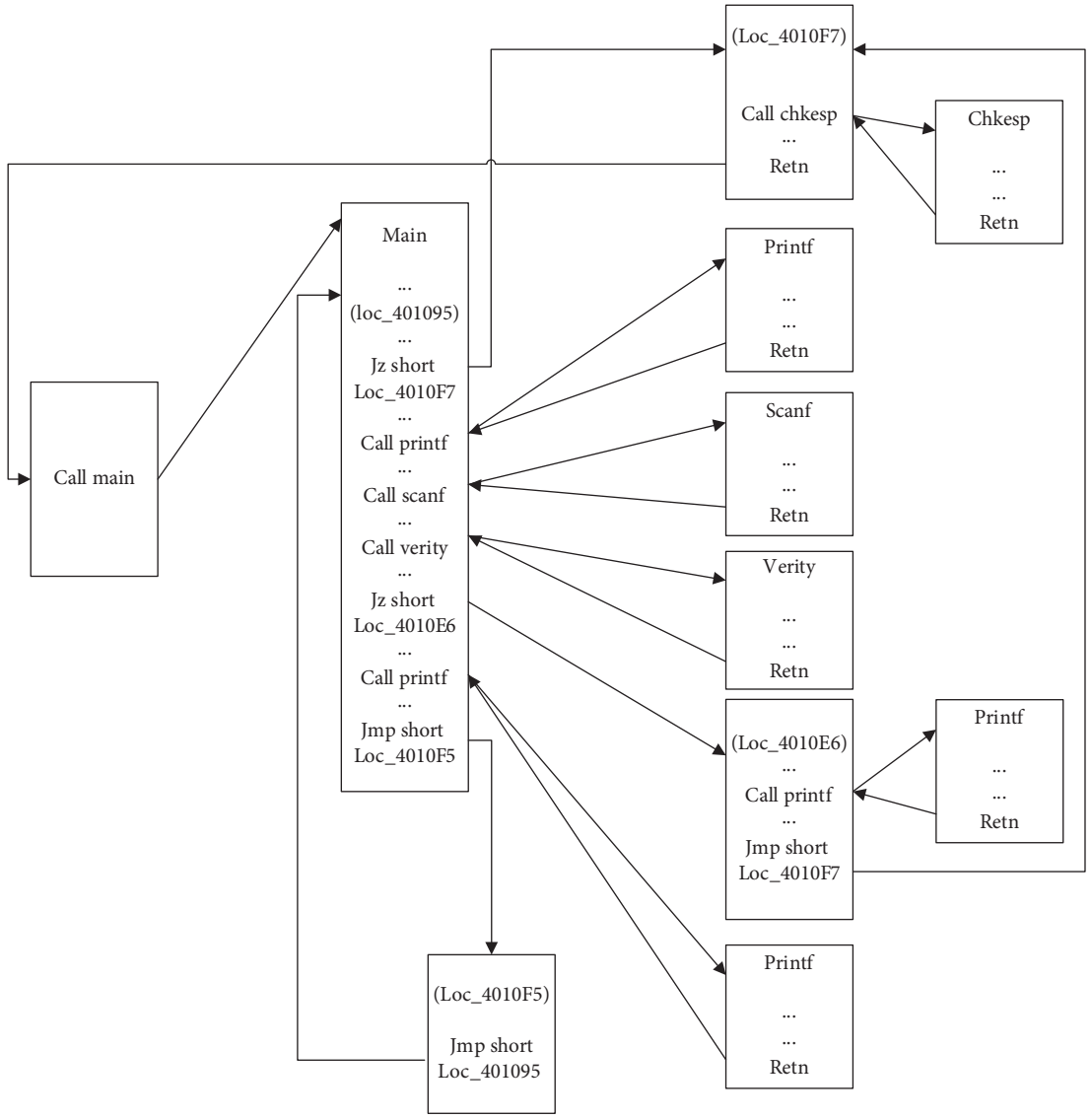

Figure 4: The flowchart for assembly code.

The CFG generation algorithm is shown below. The algorithm uses a recursive method to generate the CFG. The input is the assembly code of the exploit, and the output is the SQL file containing the CFG structure and all CFG node information. Algorithm 1 shows how to further process the statements containing key instructions in the assembly code of the exploit. Lines 7–12 are for processing the statements containing Call instructions, and lines 13–15 are for processing the statements containing Return. Lines 15–31 are to process the instruction statement containing JXX. The time-consuming is mainly on the program traversal process of the algorithm, with the time complexity $O(n)$, where n is the number of assembly code lines of the exploit.

3.3.2. Node Information Extraction Method. The node information extraction method mainly extracts the node *information* contained in the generated CFG. The instruction information in the exploit has been filtered out of a large part of the noncritical information in the process of generating the CFG. Here, the corresponding processing is mainly for the filtered information, which is convenient for use when generating the JCFG. In Figure 6, we can see the data structure design of the node attributes of the CFG. In Figure 7, we have further extracted the instruction information and subdivided it for the data structure design of the node attributes of the CFG, which are the instruction name, function name, and destination address. There are two types of command names, namely, the Call command and the JXX command mentioned.

3.3.3. JCFG Generation Method. The JCFG generated in the algorithm is recursive. The input is the CFG generated by the IDA script, and the output is the JCFG. Algorithm 2 further processes the node information of each node in the CFG. Lines 5–11 are for processing the node information of CFG nodes containing Call instructions, and lines 12–28 are for CFG nodes containing JXX instructions to process the node

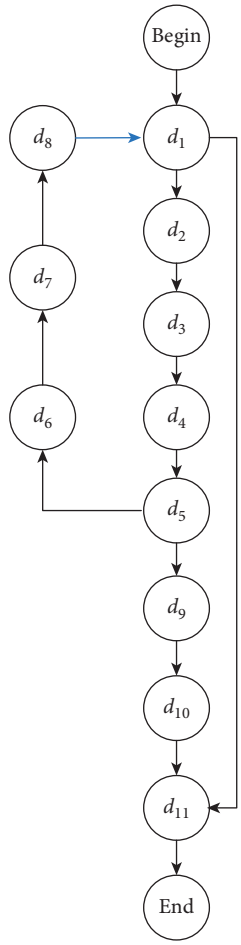

Figure 5: JCFG of a simple code segment.

```
Class CFGNode
{
    int id; // Node number
    String instruction; // Instruction information
String address; // The address of the instruction
    CFGNode *first; // Successor second node
    CFGNode *second;
}
```

Figure 6: The data structure for CFG node attribute.

information. Among them, lines 13–19 are for the node information of the CFG node of the direct jump instruction in the JXX instruction, and lines 20–27 are for the indirect jump instruction in the JXX instruction. The operation time of this algorithm is mainly spent on traversing the CFG graph nodes. As the number of CFG graph nodes is limited, the time complexity of this algorithm is $O(n)$, where n is the number of nodes in the CFG.

4. Exploit Detection Method Based on JCFG

The focus of this paper is to study the detection methods for the exploit from all aspects of the exploit. In the previous section, JCFG is proposed; in this section, we will use the JCFG to detect the exploit. This section explains the process of detecting the exploit with JCFG and the further analysis of the nodes in it with the aim of determining whether it meets the constraints of the exploit.

4.1. Exploit Detection Framework. This paper proposes a Vulnerability Exploit Detection Method based on JCFG, denoted as JCFG-VEDM, which is used to detect vulnerabilities. In addition, the proposed JCFG-VEDM is evaluated according to the detection results of this method.

Figure 8 shows the components of the JCFG-VEDM, including the following modules: JCFG generation module, execution module of exploit, and exploit judgement module.

4.2. Related Definitions and Example Analysis

4.2.1. Related Definitions

Definition 9. Function Name Judgement, CNameJudge (JCFGNode, $^*q$): when the program is dynamically executed, the current execution instruction is the Call. Comparing the function name of the node pointed to by the name of the calling function after the call and the node pointer of the current JCFG, if both are consistent, false is returned; otherwise, true is returned.

Definition 10. Jump Address Judgement, JAdressJudge (JCFGNode, $^*q$): when the program is dynamically executed, the current execution instruction is JXX, and the subsequent jump address is compared with the destination address JAdress of the node pointed to by the current JCFG node pointer. The same returns false, and the different returns true.

Definition 11. Return Address Judgement, RetnJudge (R, $*q$): when the program is dynamically executed, the current execution instruction is the Return, and the address after execution is compared with the uppermost address in the return address set R. The same returns false, and the different returns true.

Definition 12. Instruction containment: Include(d, instruction) to indicate that the currently executed instruction *node* d contains an instruction that has been defined like the Call, the JXX, or the Return. For example, $(\exists d \in \text{Prog}) \wedge (\text{Include}(d, \text{Call}))$. This means that the current execution instruction node of the Prog has Call.

Definition 13. Program execution pointer: $*q$ represents the instruction node that the program is currently executing.

4.2.2. Abnormal Jump. The execution flow is hijacked during running the program, so that the program executes code that should not be executed, which is called an abnormal jump. According to the feature definition of the exploit, $\mu(\text{AJ}) = \{D, C\}$, with a formal description of the vulnerable node D and the program feature constraint condition C of the exploit followed:

```
    Input: Exploit/ * Instruction of the exploit * /
    Output: CFG/ * CFG nodes information stored in the database * /
(1)     CFG = new CFG (); / * Initialize CFG * /
(2)     Instruction instruction; / * The command information of the current read line * /
(3)     Stack jN = new Stack <>(); / * Create a stack to store the number of instruction lines for conditional jumps and path forks
    * /
(4)     Stack R = new Stack <>(); / * Create a stack to store the address that should be returned when calling the function * /
(5)     int id = 1; / * Record the number of CFG nodes * /
(6)     for (int i = 0; i < n; i++) do
(7)       if (instruction.exist (Call)) then
(8)         CFGNode = new CFG (instruction);
(9)         if (!isSame (CFGNode)) then
(10)          CFGAdd (CFGNode);
(11)        id++;
(12)      end if;
(13)    else if (instruction.exist (Return)) then
(14)      Return (R); / * Return the address stored in R * /
(15)    else if (instruction.exist (JXX)) then
(16)      if (instruction.exist (jmp)) then
(17)        CFGNode = new CFG (instruction);
(18)        if (!isSame (CFGNode)) then
(19)          CFGAdd (CFGNode);
(20)        else
(21)          Return (jN) / * Return the address stored in jN * /
(22)        end if;
(23)      else if (instruction.exist (jnz) or instruction.exist (jz)) then
(24)        CFGNode = new CFG (instruction);
(25)        if (!isSame (CFGNode) or (isSame (CFGNode).second == null)) then
(26)          CFGAdd (CFGNode);
(27)        else
(28)          Return (jN) / * Return the address stored in jN * /
(29)          end if;
(30)        end if;
(31)    end if;
(32) end for.
```

Algorithm 1: CFG generation algorithm.

```
Class JCFGNode
{
    int id; // Node number
    NodeAttr * attr; // Node attributes
    JCFGNode *first;
    JCFGNode *second;
}
Class NodeAttr
{
    String jAddress; // Destination address
    String address; // Address of the instruction
    String attrName; // Command name
    String funcName; // Function name
}
```

Figure 7: The data structure for the JCFG node attribute.

(1) Dangerous node of exploit, $D = \{D^{\text{CNameJudge}}, D^{\text{JAdressJudge}}, D^{\text{RetnJudge}}\}$. Among them, $D^{\text{CNameJudge}} = \{d \mid (\exists d \in D) \wedge (\text{Include}(d, \text{Call}))\}$, $D^{\text{JAdressJudge}} = \{d \mid (\exists d \in D) \wedge (\text{Include}(d, \text{JXX}))\}$, $D^{\text{RetnJudge}} = \{d \mid (\exists d \in D) \wedge (\text{Include}(d, \text{Return})\}$.

Description: $D^{\text{CNameJudge}}$ represents the collection of nodes which can call the function name judgement and returns true in the program. $D^{\text{JAdressJudge}}$ represents the collection of nodes which can jump address judgement and returns true in the program. And $D^{\text{RetnJudge}}$ represents the program return address judgement and returns a collection of nodes that refer to true.

(2) Relevant constraints on exploit features, denoted as C.

$$\begin{aligned} C_1 = &\left((\exists^* p \in D, {}^*q \in \text{Prog})\right. \\ &\wedge \text{Include}\,(*p, \text{Call}) \\ &\wedge \text{Include}\,(*q, \text{Call}) \\ &\left.\wedge \text{CNameJudge}\,(*p, {}^*q)\right). \end{aligned} \tag{2}$$

Description: the constraint C_1 related to the exploit feature indicates the existence of the exploit JCFG

```
    Input: CFG
    Output: JCFG
(1)     JCFG = new JCFG (); / * Initialize JCFG * /
(2)     Stack cN = new Stack <> (); / * Create a stack to store conditional jumps and path fork nodes * /
(3)     Stack jN = new Stack <> (); / * Create a stack to store conditional jumps and path fork nodes * /
(4)     for (int i = 0; i < n; i++) do
(5)       if (node.instruction.exist (Call)) then
(6)         JCFGNode = new JCFG (nodeAttrExtract (node.instruction, node.adress));
(7)           if (JCFG.exist (JCFGNode)) then
(8)             Return (cN, jN);/ * Return the CFG node of the last forked path, and make the current JCFG node become the
    CFG node of the last forked path * /
(9)           else
(10)            JCFGAdd (JCFGNode);
(11)          end if;
(12)      else if (instruction.exist (JXX)) then
(13)        if (node.instruction.exist (jmp)) then
(14)          JCFGNode = new JCFG (nodeAttrExtract (node.instruction, node.adress));
(15)          if (JCFG.exist (JCFGNode)) then
(16)            Return (cN, jN);
(17)          else
(18)            JCFGAdd (JCFGNode);
(19)          end if;
(20)      else if (node.instruction.exist (jnz) or node.instruction.exist (jz)) then
(21)      JCFGNode = new JCFG (nodeAttrExtract (node.instruction, node.address));
(22)        if (!JCFG.exist (JCFGNode) or JCFG.second == null) then
(23)          JCFGAdd (JCFGNode);
(24)      else
(25)            Return (cN, jN);
(26)        end if;
(27)      end if;
(28)    end if;
(29) end for.
```

Algorithm 2: JCFG generation algorithm.

node $*p$ in the dangerous nodes of exploit set D and the existence of the instruction node $*q$ in the detected exploit, and the instruction of the node where $*p$ is located is Call. The instruction of the instruction node where $*q$ is located is also the Call, calling the function name judgement to determine whether there is an abnormal jump.

$$\begin{aligned} C_2 = ((&\exists\, {}^{*}p \in D, {}^{*}q \in \text{Prog}) \\ &\wedge \text{Include}\,(*p, \text{JXX}) \\ &\wedge \text{Include}\,(*q, \text{JXX}) \\ &\wedge \text{JAdressJudge}(*p, {}^{*}q)). \end{aligned} \tag{3}$$

Description: the constraint C_2 related to the features of the exploit means the existence of the exploit JCFG node $*p$ in the dangerous nodes of exploit set D, the existence of the instruction node $*q$ in the detected exploit, and the instruction of the node where $*p$ is located which is JXX. The instruction of the instruction node where $*q$ is located is also JXX. Currently, it calls the jump address judgement to determine whether there is an abnormal jump.

$$\begin{aligned} C_3 = ((&\exists\, {}^{*}p \in D, {}^{*}q \in \text{Prog}) \\ &\wedge \text{Include}\,(*p, \text{Return}) \\ &\wedge \text{Include}\,(*q, \text{Return}) \\ &\wedge \text{RetnJudge}(*p, {}^{*}q)). \end{aligned} \tag{4}$$

Description: the constraint C_3 related to the features of the exploit means that there is an exploit JCFG node $*p$ in the dangerous nodes of exploit set D and an instruction node $*q$ in the detected exploit. The instruction of the node where $*p$ is located is Return instruction, and the instruction of the instruction node where $*q$ is located is also the Return. Currently, the return address judgement is called to determine whether there is an abnormal jump.

4.2.3. Example Analysis. Figure 9 shows a code segment of an exploit. The exploiting in this program is to overwrite the return address of the strcpy function in the verity function to import the execution flow of the program into the shellcode. In the main function, the program reads the string in the password.txt and compares it with PASSWORD. The JCFG

```
Input: JCFG, The exploit node to be detected/ * The SQL containing JCFG node information and the node extracted from key
instructions of the program during execution * /
Output: Result
(1)     JCFGNode *p = JCFG.head (); / * Make the JCFG pointer point to the head node of JCFG * /
(2)     Node *q = new Node (); / * The instruction node in the execution process, the main function is first located when the
program is executed * /
(3)     Stack R = new Stack <> (); / * Create a stack to store the return address * /
(4)     vector <stack<VulNode>> Vul; / * Create a stack to record the exploit nodes that generate abnormal jumps * /
(5) begin
(6)       if (q.attrName == p.first.attr.attrName) then
(7)         if (q.attrName == Call) then
(8)           if (CNameJudge (p.first, q)) then/ *
(9)             Vul.add (temp);
(10)        enf if;
(11)      else if (q.attrName == JXX) then
(12)        if (JAdressJudge (p.first, q)) then
(13)          Vul.add (temp);
(14)        end if;
(15)      else
(16)        if (RetnJudge (R, *q)) then
(17)          Vul.add (temp);
(18)        end if;
(19)    else if (q.attrName == p.second.attr.attrName) then
(20)    else
(21)      Vul.add (temp);
(22)    end if;
(23) end
```

Algorithm 3: Vulnerability Exploit Detection Method based on JCFG (JCFG-VEDM).

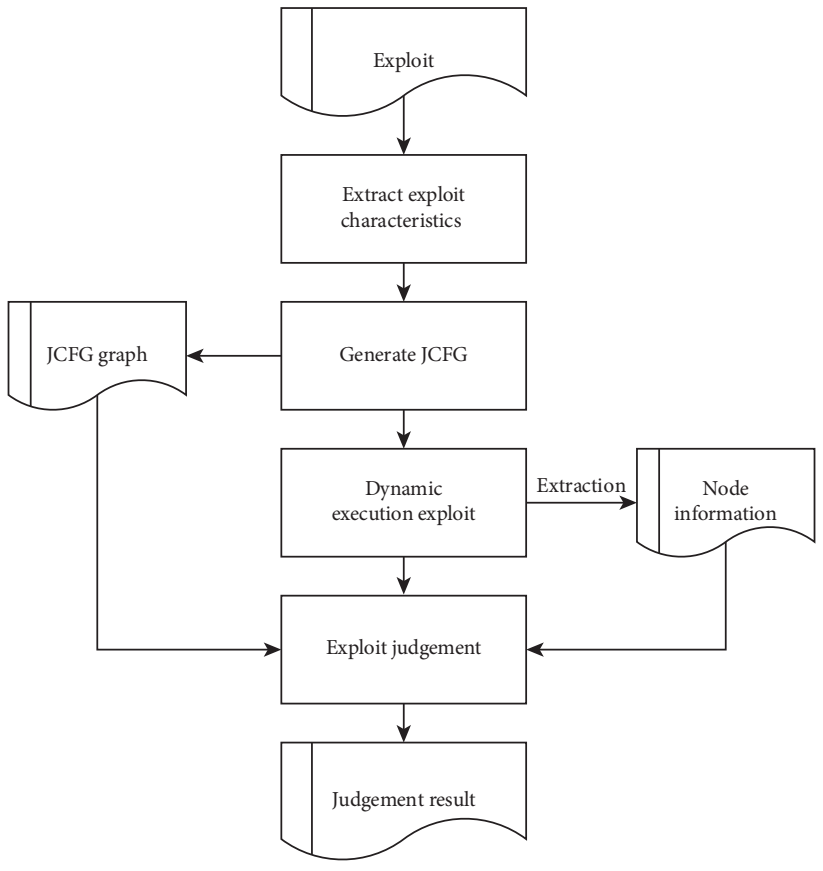

Figure 8: Exploit Detection Framework based on JCFG.

obtained by static analysis of the program is shown in Figure 10.

In Figure 10, the part of the JCFG that will cause the abnormal jump in the exploit is mainly for analysis. After the program has executed a series of instructions in the d_i node, it should return from the entered _strcpy function. The last node executed before returning is the d_j node, while the node executed afterwards should be the d_{j+1} node. The node information corresponding to these nodes is shown in Figure 11. However, when the exploit is officially executed, the return address of this program is overwritten because the program reads the information in the txt, and the execution flow is imported into the shellcode. When the program generates an abnormal jump, after the program executes the d_j node, the next instruction read is Call MessageBoxA, and the information of the $*q$ execution node is shown in Figure 11. The $*p$ node in the current JCFG is at the position of the d_j node. By comparing the node information of the subsequent nodes, the $*q$ node and the $*p$ node in the JCFG, it will be found that both do not match, with an abnormal jump generated.

4.3. Vulnerability Exploit Detection Method Based on JCFG. For the exploit to be detected, it is dynamically analyzed after static analysis is finished. Ollydbg is an extremely popular program dynamic analysis tool, through which the program can be dynamically analyzed very conveniently. When the program is dynamically analyzed, corresponding instruction nodes are generated for the key instructions in the execution process. The key instructions include the previously defined Call, JXX, and Return. The node attribute data structure design of the instruction node is shown in Figure 12. For key instructions in the execution process, the corresponding instruction nodes are generated and compared with the execution nodes in the JCFG to determine whether abnormal jumps occurred.

The specific description of the Vulnerability Exploit Detection Method based on JCFG is shown in Algorithm 3, and the part about extracting key instruction information during execution is omitted from the algorithm. The algorithm mainly shows the detection function of the exploit. According to the input execution instruction node and the node pointer of the JCFG, the detection result is obtained by matching. In lines 6–18, these are to match the subsequent first node of the execution instruction node $*q$ and the node pointer $*p$ of the JCFG. Lines 7–10 are the check function for the key instruction Call. Lines 11–14 are for the key instruction JXX check, and lines 15–18 are check functions for the key instruction Return. Then, it matches the subsequent second node of the execution instruction node $*q$ and the node pointer $*p$ of the JCFG. The specific operation is like the previous operation. Lines 20–22 indicate that the execution instruction node $*q$ does not match the subsequent first node and second node of the node pointer $*p$ of the JCFG, so the current execution instruction node and the node of the JCFG are stored in the exploit node stack. Most of the execution time of the algorithm is spent on the step-by-step reading of the execution instructions. The exploit has

```
1.  #include <stdio.h>
2.  #include <windows.h>
3.  int verity (char* password)
4.  {
5.      char buff [8];
6.      int result = strcmp (password, PASSWORD);
7.      strcpy (buff, password);
8.  return result;
9.  }
10. void main ()
11. {
12.     int flag = 0;
13.     char password [1024];
14.     FILE * fp;
15.     LoadLibrary ("user32.dll");
16.     if (! (fp = fopen ("password.txt", "rw+")))
17.     {
18.     exit (0);
19.     }
20.     fscanf (fp, "%s", password);
21.     flag = verity (password);
22.     if (flag)
23.       {
24.       printf ("false.");
25.       }
26.       else
27.       {
28.       printf ("true.");
29.        }
30.        fclose (fp);
31. }
```

Figure 9: Exploit code segment.

limited instruction statements, so the execution instruction nodes formed are less than the number of instruction statements of the exploit. The time complexity of this algorithm is $O(n)$, where n is the total number of nodes that generate and execute instructions.

5. Experimental Analysis

This chapter mainly elaborates the various information of the experiment, which includes the various indicators needed for the experiment, the prepared experimental plan, and the experimental results.

5.1. Experimental Program. This section selects some typical exploits for detection which cover a variety of attack types, including ret-to-libc, ROP, and JIT Spraying, and will give the comparison of JCFG with DEP protection strategy of the system and ASLR address randomization protection strategy. The exploits to be detected are shown in Table 1.

First, the IDA script is used to perform static analysis on the exploit, extracting the obtained assembly code, the key instruction information, and the corresponding CFG generated, which is stored in the database. Then by further extracting the node information in the CFG, the

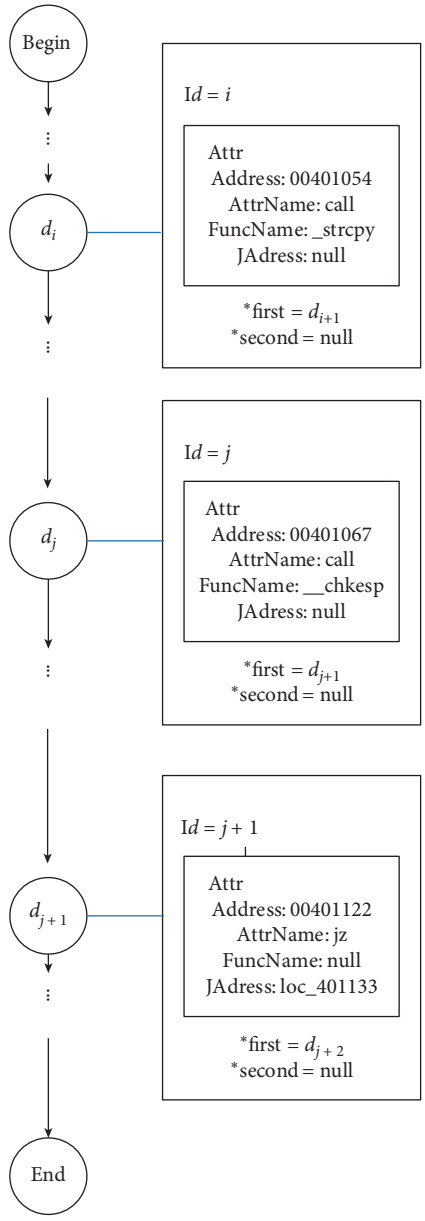

Figure 10: JCFG of the exploit.

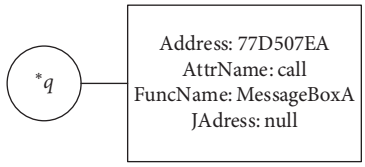

Figure 11: Program command node information during abnormal jump.

corresponding JCFG is generated and stored in the database. After obtaining the JCFG, the program is analyzed dynamically, reading the instruction information during each step-by-step execution, extracting information from the instruction information containing key nodes, and finally, obtaining the corresponding instruction node, calling the corresponding instruction determination function, and

```
Class Node
{
    String address; // Instruction address information
String attrName; // Command name
String funcName; // Instruction call function name
}
```

Figure 12: Implementation of node attribute data structure.

statistically analyzing the pointer nodes of the obtained JCFG with the node which obtain comparison to determine whether there is an abnormal jump, so as to determine whether the program to be detected belongs to an exploit.

5.2. Analysis of Results. The Vulnerability Exploit Detection Method based on JCFG can successfully detect the exploit shown in Table 1 and store the nodes that produce abnormal jumps, which are convenient for security personnel to analyze the exploit. First, the exploit is prevented by the protective measures in the system, and then the exploit is detected by the JCFG-VEDM method proposed in this paper. The detection results are shown in Table 2.

In Table 2, the detection result section uses "1" to represent that the detection method can detect the exploit and uses "0" to represent that the exploit cannot be detected. It can be seen from the results in Table 2 that neither the DEP protection strategy of the system nor the ASLR address randomization protection strategy can protect the system against the above-mentioned exploits. However, the Vulnerability Exploit Detection Method based on JCFG proposed in this paper can detect the above. It reflects that JCFG-VEDM can detect common ret-to-libc, ROP, and JIT Spraying vulnerabilities. Hence, the effectiveness and feasibility of the JCFG-VEDM detection method are verified.

Here, the detection process and detection results of the Vulnerability Exploit Detection Method based on JCFG are described.

CVE-2017-8869 is caused by a buffer overflow vulnerability in MediaCoder. Attackers can construct the .m3u file to cause the program buffer overflow and overwrite the return address of the program to execute arbitrary code. In this experiment, MediaCoder runs on the experimental host of windows 7, and the assembly code of the program is also monitored through Ollydbg. During operation, the node with ID 476 at 0x1400f92d6L shows that the successor node of this node should be the node with ID 477 in the database, but at runtime, the control flow jumps to another address. Thus, the instruction node does not match the program instruction node in the static JCFG in the generated program, and it is determined that an abnormal jump has occurred. The hacker can use it to execute the shellcode that hides in the .m3u file.

The experiment shows that the DEP and ASLR protection strategies fail to protect the system, the exploit program can execute the shellcode that hacker hided, and JCFG-VEDM can detect it by verity of the jump address of

Table 1: Vulnerability information about exploits.

CVE number	Vulnerability name	Software version
CVE-2018-9131	Reaper buffer error vulnerability	Reaper 5.78
CVE-2018-6481	Flexense Disk Savvy enterprise buffer error vulnerability	Flexense Disk Savvy enterprise 10.4.18
CVE-2017-14627	CyberLink LabelPrint buffer error vulnerability	CyberLink LabelPrint 2.5
CVE-2017-8869	MediaCoder buffer error vulnerability	MediaCoder 0.8.48.5888
CVE-2017-8870	AudioCoder buffer error vulnerability	AudioCoder 0.8.46

Table 2: The detection results of exploits.

CVE number	Detect method	Result
CVE-2018-9131	DEP	0
	ASLR	0
	JCFG-VEDM	1
CVE-2018-6481	DEP	0
	ASLR	0
	JCFG-VEDM	1
CVE-2017-14627	DEP	0
	ASLR	0
	JCFG-VEDM	1
CVE-2017-8869	DEP	0
	ASLR	0
	JCFG-VEDM	1
CVE-2017-8870	DEP	0
	ASLR	0
	JCFG-VEDM	1

executable file. And compared with the common CFI detection approach, JCFG-VEDM does not need the source code of the executable file to use the method of program instrumentation, it is convenient for security researchers to detect the vulnerability of executable program which cannot get the source code.

6. Threats to Validity

A threat to internal validity relates to the type of vulnerabilities used in the experimental analysis. To mitigate this threat, we have prepared more different CVE vulnerabilities regarding the buffer overflow vulnerability. A threat to external validity relates to the generalizability of our results because we used vulnerability data from only buffer overflow vulnerability to verify the effectiveness and the feasibility of the abnormal jump studied. Our future work will address this threat by examining other vulnerabilities like Heap Overflow, Stack Overflow, and so on.

7. Conclusion

There are certain features in the occurrence of program vulnerabilities. This paper conducts an in-depth analysis of the features of the exploits of abnormal jumps and proposes a Vulnerability Exploit Detection Method based on JCFG (JCFG-VEDM). Firstly, this method analyzes the exploit features of the exploit to obtain its corresponding JCFG. And then it uses the Ollydbg tool to dynamically analyze the exploit, generating corresponding instruction nodes for the executed key instructions. Finally, it compares nodes pointed to by the JCFG node pointer to determine whether an abnormal jump has occurred. In addition, this method also can be utilized to determine whether the program is an exploit.

To verify the effectiveness and feasibility of the JCFG-VEDM method proposed in this paper, we compare the JCFG-VEDM with the current system's DEP protection strategy and ASLR address randomization protection strategy in the experimental analysis. Experimental results show that JCFG-VEDM can detect the above-mentioned exploits, while the system's DEP and ASLR protection strategies fail to protect the system, and compared with the traditional CFI, the JCFG-VEDM does not need instrumentation, and it is minifying the workload for the security researchers to detect the exploits by using the approach.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was partly supported by the National Natural Science Foundation of China (NSFC) (Grant no. U1836116), the National Key R&D Program of China (Grant no. 2020YFB1005500), and the Leading-Edge Technology Program of Jiangsu Natural Science Foundation (Grant no. BK20202001).

References

[1] N. R. Weidler, D. Brown, and S. A. Mitchell, "Return-oriented programming on a resource constrained device," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 244–256, 2019.

[2] Y. H. Xu and Z. X. Sun, "Research development of abnormal traffic detection in software defined networking," *Journal of Software*, vol. 31, no. 1, pp. 183–207, 2020.

[3] J. Qu, C. L. Fan, G. Y. Chen et al., "Research on establishment of network security service ability system for A new era," *Netinfo Security*, vol. 19, no. 1, pp. 83–87, 2019.

[4] F. F. Wang, T. Zhang, W. G. Xu et al., "Overview of control-flow hijacking attack and defense techniques for process," *Chinese Journal of Network and Information Security*, vol. 5, no. 6, pp. 10–20, 2019.

[5] M. H. Wang, H. Yi, A. V. Bhaskar et al., "Binary code continent: finer-grained control flow integrity for stripped binaries," *Journal of Cyber Security*, vol. 1, no. 2, pp. 61–72, 2016.

[6] B. Liu, J. F. Chen, S. L. Qin et al., "An approach based on the improved SVM algorithm for identifying malware in network traffic," *Security and Communication Networks*, vol. 2021, Article ID 5518909, 14 pages, 2021.

[7] M. Abadi, M. Budiu, Ú. Erlingsson et al., "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, vol. 13, no. 1, pp. 1–40, 2009.

[8] M. Abadi, "Protection in programming-language translations," in *Proceedings of the 25th International Conference on Automata Languages and Programming (ICALP' 98)*, pp. 868–883, Aalborg, Denmark, July 1998.

[9] E. Göktas, E. Athanasopoulos, H. Bos et al., "Out of control overcoming control-flow integrity," in *Proceedings of the 2014 IEEE Symposium International Conference on Security and Privacy*, pp. 575–589, San Jose, CA, USA, May 2014.

[10] M. H. Wang, L. Y. Ying, and D. G. Feng, "Exploit detection based on illegal control flow transfers identification," *Journal on Communications*, vol. 35, no. 9, pp. 20–31, 2014.

[11] Ú. Erlingsson, M. Abadi, M. Vrable et al., "XFI: software guards for system Address spaces," in *Proceedings of the 7th Symposium International Conference on Operating Systems Design and Implementation (OSDI'06)*, pp. 75–88, Seattle, WA, USA, November 2006.

[12] R. D. Clercq, R. D. Keulenaer, B. Coppens et al., "SOFIA: software and control flow integrity architecture," *Computers & Security*, vol. 68, pp. 16–35, 2017.

[13] K. Heydemann, J. F. Lalande, and P. Berthomé, "Formally verified software countermeasures for control-flow integrity of smart card C code," *Computers & Security*, vol. 85, pp. 200–218, 2019.

[14] M. W. Zhang and R. Sekar, "Control flow and code integrity for COTS binaries," in *Proceedings of the 22nd Symposium International Conference on USENIX Security (Usenix'13)*, pp. 337–352, Washington DC, USA, August 2013.

[15] N. Dautenhahn, J. Criswell, and V. Adve, "KCoFI: complete control-flow integrity for commodity operating system kernels," in *Proceedings of the 2014 IEEE Symposium International Conference on Security and Privacy*, pp. 292–307, San Jose, CA, USA, May 2014.

[16] N. Carlin, A. Barresi, D. Wagner et al., "Control-flow bending: on the effectiveness of control-flow integrity," in *Proceedings of the 24th Symposium International Conference on USENIX Security (Usenix'15)*, pp. 161–176, Washington DC, USA, August 2015.

[17] C. Zhang, T. Wei, Z. F. Chen et al., "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE symposium International Conference on Security and Privacy*, pp. 559–573, San Francisco, CA, USA, February 2013.

[18] Y. B. Xia, Y. T. Liu, H. B. Chen et al., "CFIMon: detecting violation of control flow integrity using performance counters," in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, pp. 1–12, Washington DC, USA, June 2012.

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

[20] X. J. Wang, C. Z. Hu, R. Ma et al., "A survey of the key technology of binary program vulnerability discovery," *Netinfo Security*, vol. 17, no. 8, pp. 1–13, 2017.

[21] Z. B. Han, X. H. Li, Z. C. Xing et al., "Learning to predict severity of software vulnerability using only vulnerability description," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 125–136, Shanghai, China, September 2017.

[22] J. X. Ma, Z. J. Li, T. Zhang et al., "Taint analysis method based on offline indices of instruction trace," *Journal of Software*, vol. 28, no. 9, pp. 2388–2401, 2017.

[23] L. Wang, F. Li, L. Li et al., "Principle and practice of taint analysis," *Journal of Software*, vol. 28, no. 4, pp. 860–882, 2017.

[24] M. V. Belyaev, N. V. Shimchik, V. N. Ignatyev et al., "Comparative analysis of two approaches to static taint analysis," *Programming and Computer Software*, vol. 44, no. 6, pp. 459–466, 2018.

[25] S. Sayeed, H. Marco-Gisbert, I. Ripoll et al., "Control-flow integrity: attacks and protections," *Applied Sciences*, vol. 9, no. 20, p. 4229, 2019.

[26] P. H. Yuan, Q. K. Zeng, Y. J. Zhang et al., "Attacking web browser: ROP gadget injection by using JavaScript code blocks," *Journal of Software*, vol. 31, no. 2, pp. 247–265, 2020.