A Vulnerability Exploits Detection Method Based on Binary Program Behavior Analysis

Jinxin Zhong, Wenqing Fan, Jing An, Miao Zhang, Yixian Yang

Information Security Center, Beijing University of Posts and Telecommunications, Beijing, China

jinxin.zhong@gmail.com, fanwenqing@cuc.edu.cn, an_jing@bupt.edu.cn, zhangmiao@bupt.edu.cn, yxyang@bupt.edu.cn

Abstract

Nowadays, Vulnerability exploiting has become a major means of malicious code communication. In order to solve the problem that polymorphic and metamorphic vulnerability exploit variants is difficult to detect, it presents a method of detecting vulnerability exploits based on binary behavior analysis in this paper. The method track and monitor the memory and register changes the binary file's behavior results, and have a formal verification on application's behavior through intermediate-level language. In this paper, we examine the performance of binary analysis by taking two sets of experiments for separate targets, one is 13 local file vulnerabilities, the other is web browser vulnerabilities. The results show that the exploits detection method based on binary behavior analysis can be effectively used for analysis and detection of the vulnerabilities, also with significantly reduced time and space complexity.

Keywords: Vulnerability Exploits; Binary Program Behavior; Formal Analysis; Intermediate Language

1. Introduction

Vulnerability exploiting [1] has become a major means of malicious code communication at present. It utilizes or uses local file vulnerabilities, browser vulnerabilities and system vulnerabilities, etc. [2] to complete attacks on the user such as Trojan implanting [3], rootkit installation [4] and worm propagation[5] etc.

Now, characteristic-based detection method is mainly adopted for analysis and testing vulnerability exploiting, which determines whether vulnerability exploiting is available or not through matching known characteristics. At present, behavior-based detection methods have also been introduced into detecting vulnerability exploiting, which finish detection through monitoring the implementation of the program, and judging changes such as files, registry table, processes, etc. such as Honeypot [6-7], CaptureHPC [8], Caffeine Monkey [9] and so on. However, both feature-based and behavior-based detection methods are difficult to effectively detect vulnerability exploiting processed by polymorphism [10-11] and deformation techniques [12-13] since vulnerability exploiting processed by polymorphism and deformation techniques can prevent from being detected through methods of avoiding known checkpoint or adding confusion information to deceive detection tools. The work stage of the underlying layer of the operation system can’t be detected due the absence of research on the underlying layer of the operating system, such as changes of memory and register, only data changes and API calls of the application layer are monitored[14], the vulnerability exploiting processed by polymorphism and deformation techniques can’t be judged, and thereby it can’t be detected.

This paper presents an exploit detection method based on binary process behavioral analysis, which carries out binary process analysis through vulnerability exploiting, monitors the underlying layer of the operation system, traces change conditions of memory and registers, reveals working mechanism of the vulnerability exploiting process, and detects it, thereby effectively solving the problem that vulnerability exploiting processed by polymorphism and deformation techniques can’t be detected.
2. Formalization Description

In order to carry out binary program analysis, this paper defines an intermediate language EDIL (Exploit Detection for Intermediate Language) for security analysis. It optimizes and extends BIL intermediate language [15] aiming at detection of vulnerability exploiting. BIL is a specialized language for binary program analysis, which can correctly describe data flow and control flow of binary program.

2.1 Definition of Intermediate Language

EDIL intermediate language is defined as follows:

\[
\text{stmt} \quad ::= \text{var} \; := \text{exp} | \text{jmp}(\text{exp}) | \text{cjmp}(\text{exp}, \text{exp}, \text{exp}) | \text{halt}(\text{exp}) | \text{assert}(\text{exp})
\]

\[
\text{exp} \quad ::= \text{load}(\text{exp}, \text{exp}, \theta) \mid \text{var} \mid \overline{\text{u}} \text{exp} \mid \text{exp} \overline{\text{u}} \mid \text{store}(\text{exp}, \text{exp}, \text{exp}, \theta)
\]

\[
\text{var} \quad ::= (\text{string}, \text{id}, \theta)
\]

\[
\text{} \quad ::= +, -, \times, \div, \mod, \ll, \gg, &, |, \overline{\text{u}}
\]

\[
\text{value} \quad ::= \text{integer} \mid \text{memory} \mid \perp
\]

\[
\theta \quad ::= \theta_{\text{reg}} | \theta_{\text{mem}}
\]

\[
\text{integer} \quad ::= \text{n} (: \theta_{\text{reg}})
\]

\[
\text{memory} \quad ::= \{ \text{integer} \rightarrow \text{integer}, \text{integer} \rightarrow \text{integer}, \ldots \} (: \theta_{\text{mem}})
\]

Wherein, \(s\) represents state sentence, including the assignment sentence \(\text{var} := \text{exp}\), jump sentence \(\text{jmp}(\text{exp})\), conditional jump sentence \(\text{cjmp}(\text{exp}, \text{exp}, \text{exp})\), the termination sentence \(\text{halt}(\text{exp})\) and the assertion sentence \(\text{assert}(\text{exp})\). For example, the conditional jump sentence consists of three expressions \(e_1, e_2\) and \(e_3\). \(\text{cjmp}(e_1, e_2, e_3)\) indicates jumping from the current address \(e_3\) to new address \(e_2\) under the constraints of condition \(e_1\).

\(e\) represents the expression that describes the operation with no effect on the state, including storage, reading, binary calculation and unary arithmetic operations, etc.

Wherein, the stored expression \(\text{store}(e_1, e_2, e_3, \theta)\) represents that the value \(e_1\) with the start address of \(e_2\) is stored into memory \(e_3\), \(\theta_{\text{reg}}\) represents the stored number of bytes. Similarly, the reading expression \(\text{load}(e_1, e_2, \theta)\) represents that data are read from memory \(e_2\) to \(e_1\), and \(\theta_{\text{reg}}\) represents the stored number of bytes.

\(f\) represents a variable, and can be expressed with byte string \(\text{string}\), variable \(\text{id}\) and register \(\theta\). \(\triangleleft \text{b}\) belongs to typical binary calculations, \(\triangleleft \text{u}\) belongs to unary arithmetic operations, and is composed of three types of register value \(i\), memory value \(m\), and terminator \(\perp\).

\(\theta\) refers to the basic type structure in EDIL, which contains the register \(\theta_{\text{reg}}\) and memory \(\theta_{\text{mem}}\). Type \(\theta_{\text{reg}}\) is expressed with integer \(I\), and said, and \(n \in \{1, 8, 16, 32\}\), namely, the register types only include 1-bit, 8-bit, 16-bit and 32-bit. Type \(\theta_{\text{mem}}\) represents memory value with \(m = n_{i_1} \rightarrow n_{i_2} \rightarrow n_{i_3} \rightarrow \ldots \rightarrow n_{i_n}\), wherein \(n_{i_n}\) represents the address of the memory, and \(n_{i_1}\) represents value stored in \(n_{i_n}\), namely, the address of the memory.

2.2 Language and Grammar

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Basic Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Sigma)</td>
<td>Mapping one state sentence to another;</td>
</tr>
<tr>
<td>(\mu)</td>
<td>Mapping one memory address into current value;</td>
</tr>
<tr>
<td>(\varphi)</td>
<td>Evaluating one variable;</td>
</tr>
<tr>
<td>(\eta)</td>
<td>Program counter;</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>Next sentence to be implemented;</td>
</tr>
</tbody>
</table>
EDIL grammar consists of five parameters, namely, the program status sentence list ($\sum$), current memory state ($\mu$), current variable value ($\phi$), program counter ($\eta$) and program current state ($\sigma$), and the basic grammar of the five parameters are shown in Table 1.

In the five parameters, $\sum$, $\mu$ and $\phi$ play the mapping roles, $\phi[x \leftarrow 10]$ represents that the variable $x$ is updated, it is evaluated as value $v = 10$.

### 2.3 Formal Expression

EDIL has complete formal methods, each of its program state sentences accords with the following formal expressions:

$$\text{computation} \quad s \Rightarrow s'$$

(1)

The expression (1) indicates that when program state sentence $s$ is changed from the current state to next terminating state $s'$, other rules are changed on computation on score line. For example, When conditional jump happens successfully, the formal expression of EDIL is as the follows:

$$\mu, \phi \varphi_1 \downarrow \left\{ \begin{array}{ll} \phi \varphi_1 \downarrow & v_1 \\
\sum, \mu, \phi, \eta, c, \text{jmp}(e, e_1, e_2) \Rightarrow \sum, \mu, \phi, v_1, \sigma \end{array} \right.$$  

(2)

Formal expression (2) has a simple calculation rule to be specified. Expression $\mu, \phi \varphi_1 \downarrow v$ represents that the value $V$ of $e$ is obtained according concrete range of $\mu$ and $\phi$ under current state sentence.

### 3. method of Vulnerability Exploiting Detection

Vulnerability exploiting detection method based on binary program behavior analysis compares malicious behavior with normal behavior through static analysis on binary program behavior, thereby detecting whether binary program is a malicious vulnerability exploiting program or not. Vulnerability exploiting detection method based on binary program behavior analysis mainly includes the following three sub-algorithms: basic block partitioning algorithm, control flow analysis algorithm and behavior analysis algorithm.

#### 3.1 Basic Block Partitioning Algorithm

Basic block partitioning is carried out on the basis of assembly code, the process of the basic block can be divided into two steps:

1) To determine the entry instruction of each basic block; entry instruction of basic block only can be the following three types according to the above principles:
   - The first instruction at the entrance of program;
   - Instruction on the jump objective address of jmp or jcc;
   - Next instruction under jmp or jcc instruction.

2) The basic block can be constructed according to the entrance instructions of each basic block. Only three conditions are available here:
   - All instructions from the starting address of current entrance instruction to the starting address of next entrance instruction form one basic block;
   - All instructions from the starting address of current entrance instruction to the ending address of later first jump instruction form one basic block;
   - All instructions from the starting address of current entrance instruction to the ending of the program form one basic block.
Control Flow Analysis Algorithm

Control flow analysis is carried out on the basic of basic block partitioning aiming at simplifying the path of behavior analysis, thereby increasing the efficiency of vulnerability exploiting detection. In the paper, the process of utilizing hierarchical method [16] to analyze the control flow of the program can be divided into three rounds.

1) The first round analyzes functions in the program for obtaining the start address and end address of each function;
2) The second round calls basic block partitioning module for analyzing the assembly code in each function, and records the attributes of each basic block. The attributes with attention in the analysis program include entrance address, exit address and jump target of basic block.
3) The third round obtains all control flow paths of the function according to control flow graph, namely, all possible paths from the control flow root node (function entrance basic block) to the leaf node are obtained, and the paths are composed of all basic blocks.

Behavior Analysis Algorithm

Vulnerability exploiting detection based on binary process analysis is finished on EDIL intermediate language level, and detects whether the samples contain vulnerability exploiting or not through comparing with normal behavior description.

We record $\Omega$ as an aggregate composed of all basic blocks in EDIL intermediate language in this paper, and $\bar{F}$, an $n$ step process calling sequence is a vector in $\Omega^n$.

$$\bar{F} = (F_1, F_2, F_3, \ldots, F_n)$$  \hspace{1cm} (3)

Wherein, $F_j \in \Omega$. $\bar{F}$ refers to the sequence called by $n$ step process composed of basic blocks, and has sequentiality.

The expression evaluations of two basic blocks under the termination state are equal, $F_i$ and $F_j$ should meet the follows:

$$F_j - F_j = 0$$
$$F_i = F_j$$  \hspace{1cm} (4)

The sequence $\bar{F}$ called by $n$ step process of normal behavior description can be obtained through analyzing original process samples $P$.

$$\bar{F} = (F_1, F_2, F_3, \ldots, F_n)$$  \hspace{1cm} (5)

The sequence $\bar{F}'$ called by $n$ step process of binary process behavior description can be obtained through analyzing detection samples $P'$.

$$\bar{F}' = (F_1', F_2', F_3', \ldots, F_n')$$  \hspace{1cm} (6)

(5) and (6) are compared, only when $n = m$ and $F_i = F_i'$ ($i = 1, 2, 3, \ldots, n$), therefor $\bar{F} = \bar{F}'$.

If $\bar{F} = \bar{F}'$, the detection sample $P'$ does not have vulnerability exploiting.

If $\bar{F} \neq \bar{F}'$, the detection sample $P'$ has vulnerability exploiting. Since $\bar{F}'$ is process calling sequence, it can be traced back through $\theta$, the vulnerability exploiting points in the basic block $F_i$ can be detected.

The algorithm pseudo-code is showed Fig. 1.
4. Vulnerability Exploiting Detection Model System

4.1 Description of Model System

The Fig. 2 provides vulnerability exploiting detection model system based on binary program behavior analysis, the block diagram represents the core components, and the rest represents file formats and intermediate process. The model system includes three main parts: program behavior monitor, pre-processor and program behavior detector, and their working principles are described in details here.

Figure 2. Vulnerability exploiting detection model system based on binary program behavior analysis

4.2 Program Behavior Monitor

The program behavior monitor is used for monitoring files needing vulnerability exploiting detection, which traces the implementation conditions of the files in binary program. OllyDBG v1.1[17] is used in program behavior monitor for finishing disassembly work. OllyDBG can resolve a variety of PE formats, thereby obtaining assembly instructions, registers, memory, exporting functions and other information inside. At the same time, the program behavior monitor also can dynamically monitor the implementation path of specific course during program implementation, the adopted method is to describe Native API in descriptor table (SSDT) through services of hook system, thereby remounting the program implementation path through related stack information, binary program analysis can be assisted and simplified through monitoring calling Native API condition, especially during treatment of function calling and generation of dynamic code.

```c
BOOL compare(vector<block> F1, vector<block> F2)
{
    if(F1.size!=F2.size) return false; // F1 ≠ F2
    vector<block>::iterator ptr1=F1.begin();
    vector<block>::iterator ptr2=F2.begin();
    for(; ptr1!=F1.end(); ptr1++, ptr2++)
    {
        if(*ptr1==*ptr2)
            continue;
        else
            return false; // F1 ≠ F2
    }
    if(ptr1==F1.end())
        return true; // F1 = F2
}
```

Figure 1. Behavior analysis algorithm pseudo-code
4.3 Preprocessor

Preprocessor includes two major aspects of vulnerability exploiting detection method based on binary program behavior analysis: based on basic block partitioning and control flow analysis. The input of preprocessor is assembly language, and output is EDIL intermediate language, assembly code is also optimized when language is converted, thereby increasing the detection efficiency. For example, the assembly code of initialized distribution stacks on one program entrance point is as follows:

```
00411A20   push ebp
00411A21   mov ebp,esp
00411A23   sub esp,0E8h
00411A29   push ebx
00411A2A  push edi
00411A2B   lea edi,[ebp-0E8h]
00411A31   mov ecx,3Ah
00411A36   mov eax,0CCCCCCCC
```

The output EDIL intermediate language is as follows after conversion through preprocessor:

```
00411A20   ESP = ESP − 4;
mem = store (mem, ESP, EBX, 32);
00411A21   EBX = ESP;
00411A23   ESP = ESP−232;
00411A29   EBX = ESP − 4;
mem1 = store (mem1, EBX, EBX, 32);
00411A2A  EDI = ESP − 4;
mem2 = store (mem2, EDI, EBX, 32);
00411A2B  EDI = (EBP + 0xFFFFFFE8);
00411A31   mem = store (mem, ECX, 0E8h, 32);
00411A36   mem = store (mem, EAX, CCCCCCCCCh, 32);
```

4.4 Program Behavior Detector

The function of program behavior detector is to determine whether the detected files have vulnerability exploiting or not. It compares the file program behavior description with normal behavior description needing vulnerability exploiting detection, namely, $F^\dagger$ and $F^\ast$ are compared. The comparing method of two $n$ step process calling sequence vectors has been mentioned in section 3.3 Behavior Analysis Algorithm, we no longer give unnecessary details.

5. Experimental Results

In order to validate the effectiveness of vulnerability exploiting detection method based on binary program behavior analysis, which is proposed in the paper, file browser vulnerabilities and local file vulnerabilities are selected for respective experiments. The hardware environment CPU selected by the experiment is Core i3 530, and the memory is 4GB.

5.1 Browser Vulnerability Exploiting Detection

Experiment environment of browser vulnerability exploiting[18] detection: the operating system is WindowsXP SP3, and the browser version is Microsoft Internet Explorer (IE) v 6.0 and v 7.0. In order to ensure the comparability of test samples, the paper selects 10 popular browser vulnerabilities, the bound executable program is notepad.exe with the size of 179,712 bytes, and experimental results are as follows:
Table 2. Results of browser vulnerability exploiting detection

<table>
<thead>
<tr>
<th>Vulnerability No.</th>
<th>RISING</th>
<th>Kaspersky</th>
<th>nod32</th>
<th>Norton</th>
<th>EDIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS06013</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS06071</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS07004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS07017</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS07033</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS08078</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS09002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS10002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS10018</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS10035</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 shows the detection results of four kinds of major anti-virus software and EDIL vulnerability exploiting detection system model on 10 browser vulnerability exploiting samples, represents that it is detectable, represents it can’t be detected. The experiment results show that EDIL detection system model has the advantage of effectively detecting vulnerability exploiting samples, however, the other four kinds of anti-virus software have limitations in that the samples can’t be detected. Various testing anti-virus software can’t detect especially aiming at MS09002 and MS10018. The vulnerability exploiting mode of MS09002 is caused by the reason that IE’s CFunctionPointer function does not properly handle document object, if an object is added or deleted with specific sequence, memory corruption can be triggered, thereby leading to the condition that the authority of current login user implements arbitrary code. The anti-virus software does not have the behavior of monitoring this function, and can’t conduct detection, so the software can’t judge vulnerability exploiting after the deformation is carried out.

Table 3. Results of browser vulnerability exploiting detection

<table>
<thead>
<tr>
<th>Vulnerability No.</th>
<th>Size of Generation File (byte)</th>
<th>Size of Behavior Description File (KB)</th>
<th>Preprocessing Time (s)</th>
<th>Detection Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS06013</td>
<td>543,052</td>
<td>1,192</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>MS06071</td>
<td>543,796</td>
<td>1,045</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>MS07004</td>
<td>544,868</td>
<td>1,978</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>MS07017</td>
<td>html 124 a.jpg 181.006 b.jpg 181.006</td>
<td>2,309</td>
<td>7</td>
<td>19</td>
</tr>
<tr>
<td>MS07033</td>
<td>542,119</td>
<td>1,179</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>MS08078</td>
<td>633,387</td>
<td>4,897</td>
<td>8</td>
<td>26</td>
</tr>
<tr>
<td>MS09002</td>
<td>633,476</td>
<td>10,687</td>
<td>15</td>
<td>37</td>
</tr>
<tr>
<td>MS10002</td>
<td>html 543,489 w.jpg 2,129 GIF 43</td>
<td>36,902</td>
<td>37</td>
<td>108</td>
</tr>
<tr>
<td>MS10018</td>
<td>544,026</td>
<td>1,789</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>MS10035</td>
<td>543,819</td>
<td>3,890</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3 shows space cost occupancy condition and time cost occupancy condition of each sample during experiment. The experimental results show that the generation size of behavior description file has no relation with the size of original vulnerability exploiting file, for example, the sizes of original vulnerability exploiting files of two experiment samples of MS08078 and MS09002 are 633KB, but the sizes of behavior description files are respectively 4,897 KB and 10,687 KB because the vulnerability exploiting method of MS08078 is to build a special Web page to trigger a remote implementation code vulnerability in data binding functions, while the vulnerability exploiting method of MS09002 is that IE’s CFunctionPointer function does not correctly handle the document object. As the two vulnerability exploiting methods are different, different paths are caused during binary behavior analysis, the

129
generation size of the description files has no necessary connection with the size of original vulnerability exploiting files.

The time cost of vulnerability exploiting comprises pre-processing time and the detection time, and is proportional to the size of behavior description file. Behavior description file size is larger, the pretreatment time and the detection time are longer.

5.2 Local File Vulnerability Exploiting Detection

Experiment environment of local file vulnerability exploiting detection[19-20]: the operating system is WindowsXP SP3, the software is respectively image processing ACDSee Pro v8.1, reader Adobe Reader v9.0, image processing Adobe Photoshop v9.0 CS2 and video player QQPlayer v2.3.696.

This paper selects 13 local file vulnerabilities, the bound executable program is notepad program: notepad.exe, and the size is 179,712 bytes. The experiment results are as follows:

<table>
<thead>
<tr>
<th>Vulnerability No.</th>
<th>RISING</th>
<th>Kaspersky</th>
<th>nod32</th>
<th>Norton</th>
<th>EDIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDSee</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adobe2009-0927</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adobe2009-2990</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adobe2009-3459</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adobe2009-3953</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adobe2009-4324</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adobe2010-0188</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adobe2010-1297</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Photoshop- BMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Photoshop- PNG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QQPlayerASX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QQPlayerCUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QQPlayerSMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4 shows the detection results of four kinds of major anti-virus software and EDIL vulnerability exploiting detection system model aiming at 13 local file vulnerability exploiting. The experiment results show that EDIL detection system model can effectively detect vulnerability exploiting samples, however, the other four kinds of anti-virus software have the conditions that the samples can’t be detected and can’t detect Adobe-CVE-2010-1297. The vulnerability is caused by the reason that JBIG2 compression has buffer overflow vulnerability which can implement arbitrary Win32 code, certain space is paved in memory through utilizing JavaScript, codes are copied to the space, at the same time, the program is jumped to the appointed memory zone for implementation through inserting a specific swf file in the pdf file and covering the back address of the function during analysis of the swf files. Since the anti-virus software is not provided with the behavior characteristics of Adobe-CVE-2010-1297, the anti-virus software can’t detect it.
Table 5 shows space cost occupancy condition and time cost occupancy condition of each sample during experiment. The experimental results show that the generation size of behavior description file of local file vulnerability exploiting detection has no direct relation with the size of original vulnerability exploiting file, and the size is determined by the utilized vulnerability, meanwhile, the space and time costs are different due to different application software types, wherein the preprocessing time and detection time of QQPlayer are the shortest, the space cost occupancy condition and time cost occupancy condition of Photoshop are the maximum because the operation logic and implementation path of Photoshop are the most complicated, thereby the space cost and time cost for vulnerability exploiting detection on it are the maximum, and the consumption is the most.

6. Conclusion

The vulnerability exploiting detection method based on binary program behavior analysis can monitor the changes of memory and register through tracing the binary program behaviors, formal analysis on the program behavior can be carried out in the intermediate language level, thereby detecting vulnerability exploiting. The method has been realized in system model, the experiment result indicates that the method can effectively analyze and detect vulnerability exploiting, and has smaller time complexity and space complexity.

7. Acknowledgements

This work is supported by China High Technology Development Project (NDRC), the Fundamental Research Funds for the Central Universities (BUPT2011RC0212).

8. References

A Vulnerability Exploits Detection Method Based on Binary Program Behavior Analysis  
Jinxin Zhong, Wengqing Fan, Jing An, Miao Zhang, Yixian Yang