Against Code Injection with System Call Randomization

Zhaohui Liang\textsuperscript{1,2}, Bin Liang\textsuperscript{1,2}, Luping Li\textsuperscript{1}, Wei Chen\textsuperscript{1}, Qingqing Kang\textsuperscript{1}, Yingqin Gu\textsuperscript{1}

1. School of Information, Renmin University of China
2. Key Laboratory of Data Engineering and Knowledge Engineering, Renmin University of China, MOE
Beijing, China
lzh@ruc.edu.cn

Abstract—The existing code injection attack defense methods have some deficiencies on performance overhead and effectiveness. In order to ensure the system performance, we propose a method that uses system call randomization to counter code injection attacks based on instruction set randomization idea. An injected code would perform its actions with system calls. System call randomization on operating system level will prevent the injected code from executing correctly. Moreover, with an extended compiler, we can perform source code randomization during compiling and implement binary executable files randomization by feature matching. The experiments show that our method can effectively counter variety code injection attacks with low overhead.

Keywords—Code Injection; System Call; Randomization

I. INTRODUCTION

Code injection attacks are to exploit software vulnerabilities and inject malicious code into target program. The process control flow is modified in some way that the injected code is finally executed. In general, the term “shellcode” is used to refer to injected code.

Many techniques have been introduced to prevent code injection attacks from various angles. The most notable technique is Instruction Set Randomization (ISR) [1-4]. ISR randomizes instruction set for each process of target system, performs de-randomization before executing on CPU to recover the original instruction set and execute correctly. An attacker does not know the key of the randomization algorithm. The shellcode can’t be randomized like as the target program so that it is invalid for that de-randomized process, causing a runtime error. Code injection attack would fail to execute. ISR is showed as figure 1.

![Figure 1. Instruction Set Randomization](image)

Although ISR can effectively thwart code injection attacks, it incurs enormous performance cost because of its per-instruction de-randomization on a virtual processor and lack of hardware support [1-4]. Such a system cannot be practically deployed.

Full instruction set randomization must cause the performance to drop down quickly. To solve this problem, from the level of the OS kernel, we simply randomize and de-randomize system call of the target program and reduce the ISR overhead greatly. Moreover, using a extended compiler, we perform source code randomization during compiling and implement binary executable files randomization by feature matching.

In the rest of paper, we first present the defense principle in Section 2. We then describe the implementation in Section 3. We demonstrate effectiveness and efficiency by experiments in Section 4. We explain related work in Section 5, Finally, we conclude in Section 6.

II. DEFENSE PRINCIPLE

The majority of injected code is machine instruction, so we focus on machine instruction code injection attacks in this paper. The characteristics of the shellcode need to be noticed including (1) machine instruction complied, (2) attacking target platform oriented, (3) short code and (4) system call must be used. According to the architecture of computer system, system calls are the only interfaces for a program to access system resources. The program would fail to execute without calling system calls correctly. In essence, a shellcode would perform its actions with system calls like a normal program. Each system call has an index called system call number. OS will call the implement functions according to this number. System call number randomization on operating system level will prevent shellcode from successful execution. Our method can defeat a wide variety of code injection attacks while incurring low performance penalty.

In general, OS maintains a consistent and backward compatible mapping between system call numbers and their implement functions. First, the system call numbers of target program randomized. The original system call number $X_o$ is overwritten with a new value $X_n$ calculated by the equation (1):

$$X_n = f(X_o, r)............(1)$$

In equation (1), $f$ is our randomization algorithm, $r$ is randomization factor. There is a system call dispatcher in OS kernel which dispatches the function according to the system call number. We customize the system call dispatcher to perform de-randomization. The original system call number is recovered using the equation (2):
\[ X_n = f^{-1}(X_n, r) \] 

In equation (2) \( f^{-1} \) is our de-randomization algorithm. The target program can execute correctly. Attackers do not know that the target program has been randomized, in the kernel space our de-randomization module transforms the system call number in shellcode into another one which can not be corresponding to the implement function expected by the attacker. Finally the shellcode fails to execute because of invalid parameters or meaningless system call number. As shown in Figure 2.

![System Call Randomization and De-randomization](image)

Attacker may attempt to acquire the randomization algorithm \( f \) and randomization factor \( r \). The attempt is also defeated. First, \( f \) and \( r \) are stored in the kernel space, user-level program are unable to get them. Second, the \( f \) and \( r \) on each machine may be different. Final, we develop a dynamic scheme to enable configuration the \( f \) and \( r \) in any time.

### III. IMPLEMENTATION

We built a prototype on Linux platform, shown in Figure 3.

![The Prototype of Our System](image)

The prototype system consists of randomization, de-randomization and preprocessing.

#### A. Randomization

On one hand, a program can call system calls directly or via library functions indirectly. In this paper, the randomization to source programs will be enforced through extended GCC and GLIBC. On the other hand, the randomization to the binary executable files is also implemented.

In this paper, GCC compiler is extended to randomize source code. A program will be translated into RTL format by GCC. The main structure of RTL format is a two-way linked list which is composed by instruction nodes. Through the feature matching, the extended GCC can identify the instruction nodes containing information about system call requests. Randomization will be done on these matched instruction nodes.

The information about system call numbers has two kinds of modes. One is that the system call numbers is contained in the instruction nodes directly. For example, the system call number is 106 in the sentence \( \text{“const int 106”} \). For another one, the system call number can't be gotten from sentence directly. For example, the system call number can't be found in the sentence \( \text{“regfsi 60”} \). It only tells us that the system call number is placed in the 60th SI register.

To the former, GCC can read the system call number directly, and perform the randomization. The transformed system call number will be used to construct a \( \text{const int type rtx} \). This \( rtx \) replaces the old \( rtx \) constructed by the original system call number. To the latter, GCC will forward search the two-way linked list from the current node to find the register which contains the system call number and read the system call number, then follow the same steps as the former.

GLIBC is extended to randomize the system calls which are encapsulated in function libraries, GLIBC performs the system call mainly by means of two ways. One is \( \text{PESUDO} \), another is \( \text{INTERNAL SYSCALL} \). The randomization module will be added into these two ways respectively.

To the former, the core work is to modify the definition of \( \text{DO CALL} \). The original \( \text{“movl”} \) instruction is replaced by the other three assembly instructions. The first one is \( \text{“pushl SSYS ify(syscall name)”} \), the system call number is transferred as a parameter to the user-defined function. The second one is \( \text{“call change”, change is defined by user and performs the randomization. The third one is “addr $4,% esp”} \), it is to maintain the balance of stack when the function call returns.

To the latter, the variable \( \text{“newid”} \) is added to receive the returned value of the function \( \text{change} \). The input restriction of inline assembly is modified, the immediate number \( \text{NR_##name} \) can be replaced by the variable \( \text{“newid”} \).

The Executable and Linking Format (ELF) is a standard file format on many different platforms. The system call number can be located and rewritten by the system call instructions in ELF files.

According to our statistics, the system call can be identified by the instruction \( \text{“int 0x80”} \), and more than 98 percent system call number can be recognized by the instruction \( \text{“mov x, eax”} \). So the system call number can be obtained by feature matching and randomized by user-defined algorithm.

In addition, some parts of the data segment in ELF files contain the same assignment operations. However, they mainly
appear in the extra segment of ELF files. We can jump over the extra segment and only deal with the code segment.

B. De-randomization

A kernel module based on the Loadable Kernel Module (LKM) is design to intercept system call requests, de-randomize the system call number before the system call invoked in the kernel and store the original system call handler in the memory. Only the system call number in the target process will be de-randomized.

During the de-randomization, the inline assembly language is introduced to transfer parameters in some system calls which need be processed specially. The experiment shows that this method is effectively.

C. Preprocessing

Reducing the performance overhead is the main target of this paper. The preprocessing is employed to carry out many tasks in randomization and de-randomization.

A table can be built to maintain the mapping between the randomized system call numbers and the original system call numbers. The de-randomization of system call numbers can be accomplished quickly by looking-up the table.

Besides, the target program which is protected by our system can be configured. Usually, the security system is determined by the vulnerable area. In this paper, an interface is designed for users to select the program which need to be protected.

IV. EXPERIMENT

Our prototype system is named as CIAS. It is evaluated from two aspects.

A. Effectiveness

Some real code injection attacks are utilized to demonstrate that CIAS can effectively thwart a wide variety of code injection attacks. For example, one is that the shellcode invokes the system call ‘execve (“/bin/sh”), attackers can start a shell and execute any system command. Another is that the shellcode invokes the system call ‘root’ directly to restart the computer. CIAS can defeat these attacks successfully.

B. Efficiency

The physical test platform is Linux 2.4.20-8 with 2.40GHz Intel Pentium IV processor, 256M RAM and GCC 3.2.2. There are three experiments as following:

First, we measured some single system call such as getpid , sethostname and open. In comparison with, we tested the add . The results are shown in Table I.

<table>
<thead>
<tr>
<th></th>
<th>getpid</th>
<th>sethostname</th>
<th>open</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time without CIAS (μ sec)</td>
<td>0.00285</td>
<td>0.470</td>
<td>0.516</td>
</tr>
<tr>
<td>Time with CIAS (μ sec)</td>
<td>0.00285</td>
<td>0.532</td>
<td>0.579</td>
</tr>
<tr>
<td>Overhead</td>
<td>0.00%</td>
<td>13.19%</td>
<td>12.21%</td>
</tr>
</tbody>
</table>

Table I indicates that the performance of add is not affected by CAIS. The reason is that there is no system call in add. The overhead of getpid is the largest one which is 13.19 percent. It is because the increase in runtime caused by CAIS is constant for each system call. getpid is a lightweight system call, its runtime is less, so that its overhead is obvious. Even so, the 13.19 percent overhead is still accepted from the view of getpid . open is a complex system call, its runtime is longer, its overhead is lower.

Second, we measured some commands which contain several system calls such as tar, gzip, cp and GCC commands. The results are shown in Table II.

<table>
<thead>
<tr>
<th></th>
<th>tar</th>
<th>gzip</th>
<th>cp</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time without CIAS (sec)</td>
<td>1.61</td>
<td>8.37</td>
<td>0.38</td>
<td>11.48</td>
</tr>
<tr>
<td>Time with CIAS (sec)</td>
<td>1.67</td>
<td>8.60</td>
<td>0.62</td>
<td>11.51</td>
</tr>
<tr>
<td>Overhead</td>
<td>3.73%</td>
<td>0.35%</td>
<td>6.90%</td>
<td>0.26%</td>
</tr>
</tbody>
</table>

By comparison with Table I, Table II shows that the overhead of commands is much low. The reason is that a command is more complicated than a system call. Besides system calls, a command also contains some time-consuming operations.

Third, we used UnixBench (Version 4.0.1) to measure the system performance before and after loaded CIAS. The results are shown in Table III.

<table>
<thead>
<tr>
<th>TEST BASELINE RESULT INDEX</th>
<th>TEST BASELINE RESULT INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Test (type = double)</td>
<td>29820.0 529258.9 177.5</td>
</tr>
<tr>
<td>Dhrystone 2 using register variables</td>
<td>116700.0 3472143.9 297.5</td>
</tr>
<tr>
<td>Exec Throughput</td>
<td>43.0 3241.4 753.8</td>
</tr>
<tr>
<td>File Copy 1024 bufsize 2000 maxblocks</td>
<td>3960.0 207819.0 524.8</td>
</tr>
<tr>
<td>File Copy 256 bufsize 500 maxblocks</td>
<td>1655.0 82188.0 496.6</td>
</tr>
<tr>
<td>File Copy 4096 bufsize 8000 maxblocks</td>
<td>5800.0 278457.0 480.1</td>
</tr>
<tr>
<td>Pipe Throughput</td>
<td>12440.0 668401.1 537.3</td>
</tr>
<tr>
<td>Process Creation</td>
<td>126.0 10498.9 833.2</td>
</tr>
<tr>
<td>Shell Scripts (8 concurrent)</td>
<td>6.0 101.0 168.3</td>
</tr>
<tr>
<td>System Call Overhead</td>
<td>15000.0 384542.1 256.4</td>
</tr>
<tr>
<td>FINAL SCORE</td>
<td>396.6</td>
</tr>
</tbody>
</table>

Table III shows that the majority of test items don’t decrease obviously except “System Call Overhead” and “Process Creation” which have a few decrease of 2.72% and 2.43% respectively. The reason is that the both items contain many system calls. However, according the FINAL SCORE, the total performance only decreases 0.74%.

All of these demonstrate that CIAS is high performance.
V. RELATED WORK

Address Space Layout Randomization (ASLR) is a technique proposed by the PaX \cite{5} team to prevent code injection attack. It has been deployed by many operating systems such as Linux kernel 2.6 and Windows Vista. However, ASLR suffers from a number attacks. Michal Bucko from HACKPL Security Lab \cite{6} pointed out that some attack techniques such as Heap Spraying could bypass ASLR. RandSys \cite{7} implemented randomization on system call level and used DES algorithm to encrypt important data. But lack of stability is the most serious disadvantage as the result of its modification of the kernel code of Linux. StackGuard \cite{8} tried to randomize on multiple objects, but it is not easy to use since the kernel of Linux need to be recompiled and the performance overhead is very high. According to Monica Chew \cite{9}, the cost of StackGuard is up to 30%. Yoshihiro Oyama \cite{10} enhanced the system performance and usability by using kernel modules. He encrypted system call arguments with XOR operation and random numbers which are not security. In additional, his approach can’t deal with some situations such as system calls directly appeared in source code and in binary executable files without source code.

VI. CONCLUSION

We described our randomization scheme on the level of OS kernel to counter code injection attacks. We only randomize and de-randomize the system call numbers of the target program which is configurable by users. The pre-processing is introduced to enhance the system performance. The de-randomization can be accomplished quickly by looking-up the table which has been built in preprocessing. Besides, we implement source code randomization by extended compiler and binary executable files by feature matching. The experiments show that our prototype system can effectively thwart a great deal of code injection attacks with low overhead.

REFERENCES