A NEW MECHANISM FOR OS SECURITY
Selective Checking of Shared Library Calls for Security

Dae-won Kim, Geun-tae Bae
Electronics and Telecommunications Research Institute in Korea

Yang-woo Roh, Dae-yeon Park
Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology

Keywords: Security attacks, OS security, Dynamic program loader, Software vulnerabilities, Kernel 2.6, Global Offset Table, Procedure Linkage Table

Abstract: This paper presents a systematic solution to the serious problem of GOT/PLT exploitation attacks. A large class of security mechanisms has been defeated by those attacks. While some security mechanisms are concerned with preventing GOT/PLT exploitation attacks, however, they are not complete against GOT/PLT exploitation attacks or the considerable performance decline occurs. We describe the selective checking of shared library calls, called SCC. The SCC dynamically relocates a program's Global Offset Table (GOT) and checks whether the accesses via Procedure Linkage Table (PLT) are legal. The SCC is implemented by modifying only the Linux dynamic loader, hence it is transparent to applications and easily deployable. In experiment results, we show that the SCC is effective in defeating against GOT/PLT exploitation attacks and is the mechanism with the very low runtime overhead.

1 INTRODUCTION

The C and C++ languages are popular primarily because of the sensitive control they provide over system resources including memory. This control is more than most programmers can handle, as appeared by the memory-related programming errors which torment programs written in these languages. Attacks which exploit memory errors such as buffer overflows constitute over the 60 percentage of serious attacks reported by organizations such as the CERT Coordination Center, and are concerned with important threats to the computing environment.

A number of attacks which exploit memory vulnerabilities have been developed. The earliest of these to achieve widespread popularity was the stack smashing attacks (Aleph One 2000, Mudge 1997), in which a stack buffer is overflowed so that a return address stored in the stack is overwritten with the starting address of injected shellcode. (See Figure 1). To avoid such attacks, several approaches were developed, which, in one way or another, prevent undetected modifications to a function's return address. They include the StackGuard (Crispin 1998) of putting canary values around the return address, so that the stack smashing can be detected when the canary value is contaminated; and others (Arash 2000, Tzi-cker 2001). Despite numerous technologies designed to prevent *buffer overflow* vulnerabilities, the problem persists, and the buffer overflows remain the dominant attacks of software security vulnerabilities.

Attacks have moved from stack smashes (Aleph One 2000) to heap overflows (Michel 2001), format string vulnerabilities (Crispin 2001), multiple free errors (Anonymous 2001), *return-into-library* (Rafal 2001), etc. which bypass existing buffer overflow defences such as StackGuard (Crispin 1998), LibSafe (Arash 2000) and non-executable memory segments (Solar Designer). While mechanisms to collapse these attacks are effective in protecting a system against the specific attack they focus on, incorporating many individual techniques to defend against a wide range of attacks is nontrivial and often requires resolving conflicting requirements imposed by the different techniques. Many new defence mechanisms to prevent new attacks lead us to conclude that additional ways to exploit the

Kim D., Bae G., Roh Y. and Park D. (2005).
A NEW MECHANISM FOR OS SECURITY - Selective Checking of Shared Library Calls for Security.
In *Proceedings of the First International Conference on Web Information Systems and Technologies*, pages 381-388
DOI: 10.5220/0001226503810388

```
void authenticate (char *argv)
{
  char buf[128];

  strcpy (buf, argv);
  printf ("* %s key is authenticated.\n", buf);

  return;
}
```

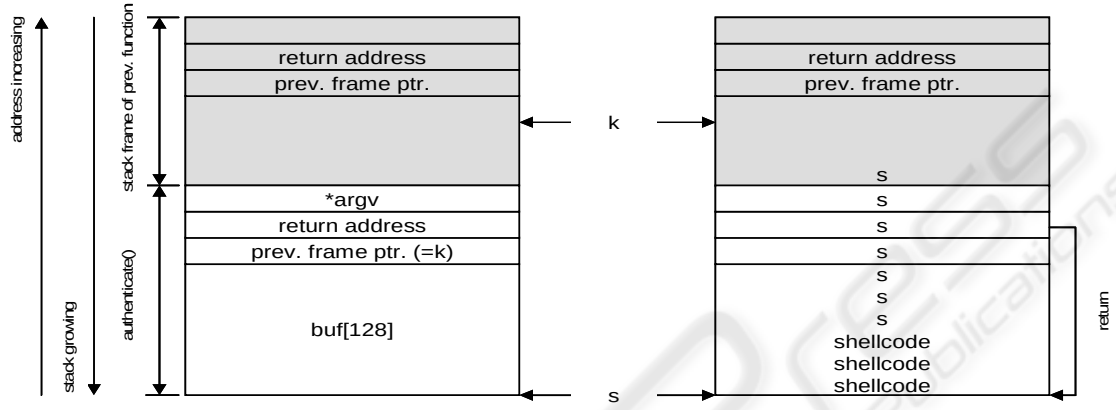

Figure 1: The stack smashing attack.

memory vulnerabilities of C and C++ will continue to be emerged in the future. As a first step towards developing more general solutions against memory exploits, we observe that an attacker must correctly determine the runtime address values of the control information position such as the return address and the address where the malicious code is located.

This paper proposes, the Selective Checking of shared library Calls (SCC), a generalized approach to protect systems against GOT/PLT exploitation attacks that exploit memory vulnerabilities. The Linux Kernel 2.6 dynamically and randomly relocates a program's stack, heap, shared libraries, and makes the stack and heap non-executable. (See Figure 2). To totally overcome a number of software vulnerabilities, these security mechanisms will be adapted. The SCC mechanism can support *memory layout randomization* mechanisms such as the Linux Kernel 2.6. In addition to the Linux Kernel 2.6 security mechanisms, the SCC dynamically relocates a program's Global Offset Table (GOT) and checks whether the accesses via Procedure Linkage Table (PLT) are legal. Making a program's GOT position different each time it obfuscates the attacker's assumptions about the addresses of GOT entries of the vulnerable program and makes the determination of critical address values difficult if not impossible. Checking the accesses via PLT frustrates the trial of attacker to illegally call shared libraries such as *system* with malicious argument ("*/bin/bash*").

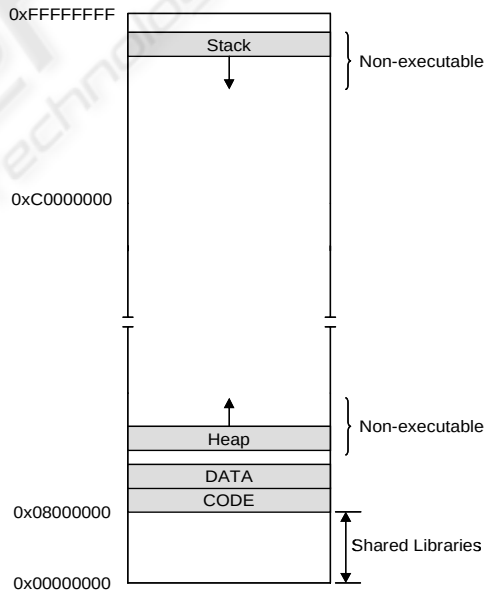

Figure 2: The linux kernel 2.6 address space.

In this paper, the PLT checking is the main contribution of SCC. The SCC is implemented by modifying the dynamic program loader compatible with Linux Kernel 2.6.x, therefore, it is transparent to the application programs, i.e., existing applications run without any modification or recompilation. To date, the SCC has been implemented on Linux Kernel 2.6.x/IA-32

platforms. It is shown to be effective against GOT/PLT exploitation attacks, and has the low runtime overhead.
The rest of this paper is organized as follows. In Section 2, we describe several related works, and Section 3 introduces the motivation and detailed mechanisms of our idea, followed by the analysis of experiments in Section 4. Finally, Section 5 provides conclusion.

2 RELATED WORKS

Address randomizing is an instance of the broader idea of introducing diversity in nonfunctional aspects of software, and idea suggested by Forrest, Somayaji, and Ackley (1997). Their implementation model was called a randomizing compiler, which can introduce randomness in several non-functional aspects of the compiled code without affecting the language semantics. As a proof of concept, they developed a modification to the gcc compiler to add a random amount of *padding* to each stack allocation request. This transformation defeats most stack smashing attacks prevalent today, but does not work against the overflow attacks with a large amount of *NOPs*.

The PaX project has developed an approach for randomizing the memory regions occupied by a program code and data, called Address Space Layout Randomization (ASLR). It have modified the Linux Kernel so that it randomizes the base address of different segments of memory, such as the stack, heap, code, and mapped shared library segments. There are, however, several weak features to ASLR. ASLR requires changes to the Linux Kernel. Kernel-level implementation requires re-installation or even reboot of the operating system. While the GOT is a frequent target of many attacks, ASLR doesn't randomize the location of GOT in the SEGMEXEC mechanism on i386. The performance impact of ASLR about the PAGEEXEC based on the fault mechanism is not yet to be officially evaluated. Finally, the implementation and detailed mechanisms are seriously architecture-dependent.

Xu, Kalbarczyk, and Iyer developed transparent runtime randomization (TRR) (Jun 2003), in which the dynamic loader is modified to randomize the base address of stack, heap, dynamically loaded libraries, and GOT. This mechanism, however, doesn't consider *return-into-PLT* (Nergal 2001) attacks. To allow return-into-PLT attacks can't be concerned with the complete mechanism for preventing the illegal operations of shared libraries.

3 SELECTIVE CHECKING OF SHARED LIBRARY CALLS (SCC)

3.1 Motivation

While TRR (Jun 2003) has the low initialization overhead and no runtime overhead, it is imperfect against return-into-PLT (Nergal 2001) attacks. The PaX project doesn't randomize the location of GOT or may allow many fault handling overheads due to the CODE region relocation to prevent return-into-PLT attacks. We observe that there are a common characteristic of return-into-PLT attacks. That is the fact that the number of libraries selected by attackers to get the critical authorities (*root* or *administrator*) of target system is a few of hundreds and thousands of shared libraries. The shared libraries they require to attack are system calls such as *execve*, *system*, setuid32, chmod, etc. Other shared libraries are not appropriate to accomplish the purposes of attackers, *root* shell acquisition and so on.

To exploit critical system calls, attackers can call normal libraries that include these system calls. While return-into-PLT can be exploited by attackers if there are buffer overflow vulnerabilities in program code, the way to detour normal libraries is very difficult to find some libraries satisfying some attack requirements (e.g. *system ("/bin/bash")* included in normal libraries.) and to manipulate some arguments of critical system calls in memory layout randomization mechanisms.

In environments of the stack and heap non-executable such as Linux Kernel 2.6, if attackers can't run the inserted shellcode to acquire root authority through overflowing buffers, those overflows don't mean serious attacks. We arrived in one conclusion by the facts that to relocate the CODE region to prevent return-into-PLT generates continuously some fault handling overheads and the number of shared libraries required to succeed return-into-PLT attacks is within the limit of a few libraries.

The SCC, our mechanism, is designed through these facts. The SCC relocates Global Offset Table (GOT) through a similar idea to TRR (Jun 2003) and checks whether the accesses via Procedure Linkage Table (PLT) to call shared libraries such as above system calls are legal. The '*legal*' means that the shared libraries are called from the *call* instruction of CODE region. All PLT accesses except for those are '*illegal*'.

3.2 The Operations of SCC

The objectives of SCC are to randomize the GOT location and check whether the accesses via the specified PLT entries are legal. The GOT relocation is similar to TRR (Jun 2003), and the PLT checking can be achieved using PLT rewritings and inserting the Checking Code in a random memory space. Both the GOT relocation and the PLT checking mechanisms are only dynamic loader modification approach, not Kernel.

Figure 3 shows the typical sequence of steps required to launch an application, using '*vim*' as an example. In this example, a user types 'vim' at the shell prompt, and the shell creates a child process using the *fork* system call. The new child process uses the *execve* system call to load and initialize 'vim'. Inside the execve system call, the operating system kernel maps the executable into memory, sets up its CODE/DATA segments, stack, heap and dynamic program loader, and then transfers the program control, which is the program counter (*%eip*), to the dynamic program loader. The dynamic program loader maps the shared libraries required by 'vim' into memory. Finally, the dynamic program loader hands over the program control to the entry point of 'vim', and 'vim' begins to execute. The SCC operations are shown in 2, 3 and 4 of 'Dynamic program loader'.

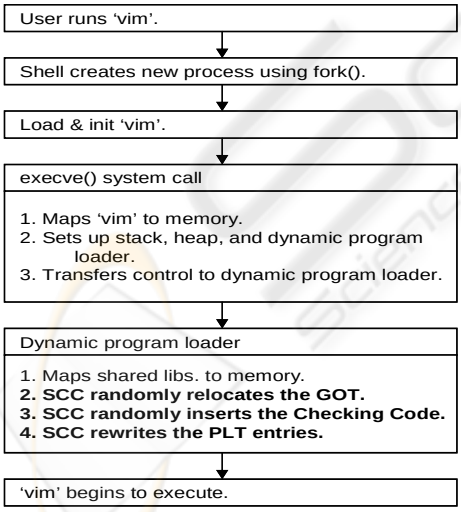

Figure 3: The SCC operations.

3.3 The Overview of SCC

It is assume that the GOT is already randomized by the dynamic program loader through a similar mechanism of TRR (Jun 2003). The PLT, therefore, must be rewritten to correctly refer to new GOT.

In Figure 4, the PLT can be accessed by the return of current function when the return address in stack is overwritten by a attacker (ATTACK), or can be accessed by the legal call due to the operation of call instructions in the CODE region. As mentioned earlier, attackers are primarily concerned with a few of critical system calls.

We modify the PLT entries (in Figure 4, PLT2) which a attacker requires to get root shell. The each entries of PLT are related with each shared library functions, and when a PLT entry (PLT2) modified for security is accessed, a changed jmp instruction in the PLT entry (PLT2) passes the program control to the Checking Code. The Checking Code checks whether this PLT access is legal, and if 'legal', the Checking Code passes the program control to the related shared library with the reference of address value in the GOT. In the case of other PLT entries, only one of three instructions in each original PLT entries is modified. The accesses to other PLT entries, therefore, are operated like as no SCC.

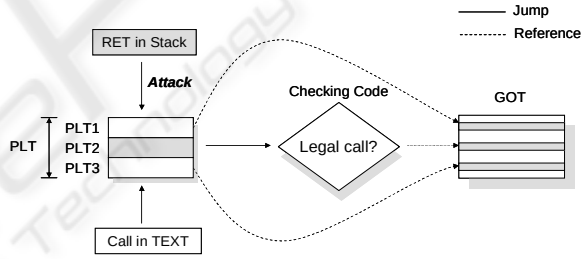

Figure 4: The overview of SCC.

3.4 The Checking Code

Figure 5 shows the stack status when the PLT entry is just accessed by the *legal call* in the CODE region and the illegal return in the stack. The PLT2 entry of Figure 4 assumes the entry related with a critical library function such as *system* and the entry that instructions have been modified by our dynamic program loader. The main idea of Checking Code is that the return address value is still remained in the stack after returning due to the return address overflowed by a attacker. If the return value in stack is equal to the accessed PLT entry address, this PLT access can be determined as the attack trial. In Figure 5, SP is the stack pointer and the things that two thunder marks are pointing present the contents in (SP-4) address. The Checking Code is mapped to the random position in the shared library region by the dynamic program loader.

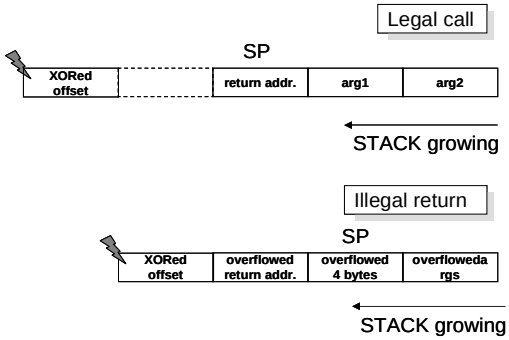

Figure 5: The stack status when the critical PLT entries are just accessed.

The Checking Code requires two values to check whether the critical PLT entry is accessed by attacks or not. The first value is the content of (SP-4) address when the PLT entry is just accessed. In Figure 5, that is the content of a thunder-marked word, and if this value is related with the accessed PLT address, the PLT entry access is determined as the attack trial. If legal call, this value is not related with the accessed PLT entry. The second value required for the Checking Code is the address of accessed PLT entry. It is used to compare with the content of (SP-4) address.

In the case of real SCC's mechanism, the accessed PLT entry inserts the XORed offset, (it is exclusive OR of original offset in the PLT entry and a random-generated value during the initial phase of dynamic loader.), value in the stack to help the Checking Code to identify the accessed PLT entry address and to defend possibly the other types of return-into-PLT attacks. There are two types of return-into-PLT attacks. In the case of first attack type, the program control is directly changed to the PLT entry (PLT2) by using the overflowed return address from the stack. The second attack type overflows the stack with the offset of critical PLT entry and returns to the *PLT_init*. (See Figure 8). To prevent this second attack type, XORing the offset is required.

Figure 6 shows the stack status including a XORed offset inserted by the modified PLT entry when the Checking Code is just accessed. The Checking Code can use the values required to check attacks because the overflowed return address and the accessed PLT entry address (be calculated by XORed offset) are in the stack.

Figure 7 shows the operations of Checking Code. In (1), the original offset is calculated by *XOR (the XORed offset)*. We can calculate the GOT entry address related with the original offset because there are a regular rule between the original offset and GOT entry address. In (2), the GOT entries related with the critical PLT entries are initialized as zero when mapping the new GOT. In the case of *lazy-loading*, if a shared library is first called, the dynamic program loader resolves a called library address and writes the address to the desired GOT entry. If the value in the GOT entry is zero, the Checking Code transfers the program control to the dynamic program loader to resolve the address of a desired shared library. If not, the Checking Code passes the program control to the related shared library with the reference of address value in the GOT.

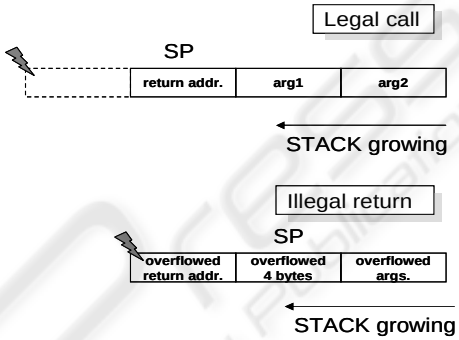

Figure 6: The stack status when the checking code is just accessed.

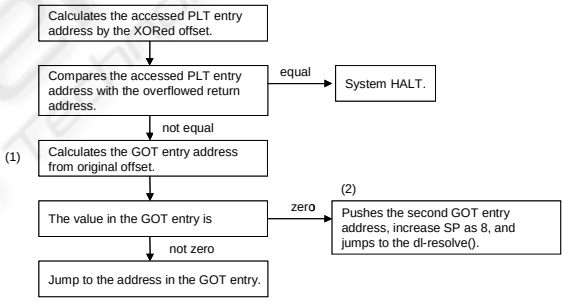

Figure 7: The operations of checking code.

3.5 The PLT Entry Modification

We explain how the selected PLT entries are modified. Figure 8(a) shows the original PLT entries in the CODE region. Each PLT entry consists of three instructions and the PLT entries (In Figure 8, PLT2) to be selected for security are the entries jumping to the critical shared libraries such as *system*. In Figure 8(b), the dynamic program loader rewrites PLT entries for pointing at new GOT entries and jumping to the Checking Code when the critical PLT entries (PLT2) are accessed. The dynamic program loader sets the writable flag of CODE region and rewrites the PLT entries from *PLT_init* to the end of PLT. The all GOT_entry

values of Figure 8(b) are new GOT_entry values. The dynamic program loader can find out the real names of shared libraries by using *fixup* function of dynamic program loader referring to each offset

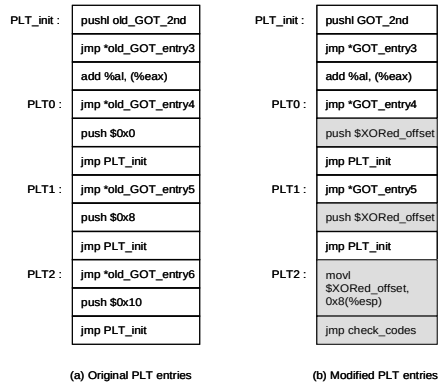

Figure 8: The original and modified PLT entries.

values, $0x0 and so on, and can search what are the critical PLT entries. We assume the PLT2 entry is a critical PLT entry. The information required for the Checking Code is a return address in the stack and the accessed PLT address. The purpose of PLT modification is to give this information to the Checking Code.

In the PLT2 entry of Figure 8(b), the first instruct-ion inserts the XORed_offset value in (SP-8) address to maintain the malicious return address of attacker. The accessed PLT address can be calculated from the XORed_offset value. The second instruction passes the program control to the Checking Code.

4 EFFECTIVENESS AND PERFORMANCE EVALUATION

This section describes the experimental evaluation of SCC. Subsection 4.1 describes the SCC's effectiveness against the attacks related with the GOT and PLT. Subsection 4.2 describes the performance cost of SCC through various types of programs. In this experiments, execve(), execl(), execlp(), execle(), execv(), execvp(), chmod(), setuid32(), chown32(), setresuid32(), fchown32(), fchmod(), setpgid() and system(), these 14 system calls are checked from the PLT entry accesses due to attacks. The measurement is taken on a PC with Kore Linux 2004 (Kernel 2.6.7), Pentium III 800MHz processor, 256MB memory and dynamic program loader ld-2.3.3.so we modified.

4.1 Effectiveness Evaluation

Here we illustrate the SCC's effectiveness in thwarting attacks related with the GOT and PLT. The effectiveness of SCC was tested using publicly available vulnerable programs and attacks against them. The programs listed in Table 1 are conventionally installed as *SetUID* root. If the attacker can get on of these programs to start a shell, then the attacker gets a root shell. The vulnerabilities and the attacks we used are presented below.

Table 1: Evaluation against security attacks.

Program	Description	No SCC	SCC
plt-exploit1	stack overflow /PLT	local root shell	detected
plt-exploit2	stack overflow /PLT	local root shell	crash
null httpd	heap overflow /GOT	remote root shell	crash
sendmail	integer overflow /GOT	local root shell	crash

- We made a simple stack overflow program, *plt-exploit1.c*, related with the return-into-PLT attack. plt-exploit1.c is the attack directly returning to the critical PLT entry. When a large number of strings are supplied to the program by a attacker, the stack buffer is overflowed, and when the vulnerable function is returned, the program control is moved to the critical PLT entry, and a *root* shell is created.
- We made second simple stack overflow program, *plt-exploit2.c*, related with the return-into-PLT attack. When the vulnerable function is returned, the program control is returned to the start address of PLT_init, and a root shell is created. The details of mechanism are described to the Phrack document (Nergal 2001).
- *null httpd* is a web server for Linux. A heap overflow vulnerability exists in its handling of the POST request. The attack passes a negative content length to start a heap overflow, overwrite a function pointer in the GOT, and create a remote root shell.
- *sendmail* is the email agent that sends messages to remote hosts. An integer overflow vulnerability exists in sendmail's function when it uses user-supplied signed integer to address an array. The attack uses a large number to overwrite a function pointer in the GOT to create a local root shell.

This is not a comprehensive exploit list, but it showed that all tests are terminated with the crash or detection message. They didn't invoke a root shell.

4.2 Runtime Overhead

The runtime overhead occurs when the Checking Code is accessed, and if the critical PLT entries are accessed the Checking Code is processed. The overhead size is, therefore, dependent to the overhead of Checking Code itself and the accessed number of modified PLT entries. The Checking Code may be constructed with 15 line assembly codes, and the average processing time of Checking Code is about 0.072us (The avg. of 1000 runs). The overhead of Checking Code itself is, therefore, very small because one instruction is processed within one cycle to the pipelined-architecture.

Table 2: The count of used system calls.

Program	The Count of System Calls		
	Total	Critical	Description
traceroute	918	1	setuid32(1)
hanterm	831	4	chown32(1) chmod(1) setresuid32(2)
passwd	1113	2	fchown32(1) fchmod(1)
emacs	21086	2	chmod(1) setpgid(1)
vim	689	2	chmod(1)chown32(1)
/bin/bash	928	2	setpgid(2)
mozilla	43088	3	execve(1) chmod(2)
telnet	514	0	

Table 2 shows the count of system calls from the program start to the program end. The '*strace*' program in Linux can count the number of used system calls. The above four programs are *SetUID* programs and the below four programs are normally popular programs. In these results, we can know the accessed number of Checking Code is even smaller than the accessed number of total system calls. Although the 'critical' number of some programs increases according to the running time or other versions of same program, the increasing number is relatively very smaller than that of total system calls.

Table 3 shows the elapsed time of pure code running except for the time waiting for some user inputs, etc. The evaluation program is supported as '*/usr/bin/time*' in Linux. When a specified program finishes, '*/usr/bin/time*' writes a message to standard out giving timing statistics about the program. Because the resolution of '*/usr/bin/time*' is millisecond (ms) unit, we consider the result of '0s 024' as the result of '24000 us'. The 'SCC overhead' means the total overhead time due to the accessed Checking Code. We measured the number of clock cycles and convert them to microseconds using the processor clock frequency. These experiments show that the runtime overhead generated by the SCC is very small. (nearly 0%).

Table 3: The runtime overhead.

Program	The Elapsed Time (usec)		
	No SCC	SCC overhead	Overhead (%)
traceroute	24000	0.5821	0.0024
hanterm	308000	1.356	4.403×10^{-4}
passwd	29000	0.662	0.0023
emacs	994000	1.17	1.177×10^{-4}
vim	107000	0.882	8.243×10^{-4}
/bin/bash	128000	0.876	6.800×10^{-4}
mozilla	3169000	1.18	3.724×10^{-4}
telnet	16000	0.312	0.195×10^{-4}

5 CONCLUSION

This paper proposes Selective Checking of shared library Calls (SCC) against GOT/PLT exploitation attacks. The underlying principle of SCC follows to randomize the application memory layout so that it is virtually impossible to determine locations of critical program data such as buffers, return addresses, and pointers of function. New idea against return-into-PLT attacks is applied to the SCC because the randomization of CODE region produces continuously many runtime overheads. The SCC's mechanism is fully transparent to application programs because it is implemented by modifying only the dynamic program loader. The effectiveness of SCC shows that it can defeat some known GOT/PLT attacks. Performance measurements show that the SCC produces very low runtime overhead.

REFERENCES

"Aleph One", 2000. The Stack for Fun And Profit. *Phrack 14(49).*

Anonymous, 2001.Once Upon a Free(). *Phrack 9(57).*

Arash, B., Navjot, S., and Timothy, T., 2000. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00),* pages 251-262, Berkeley, CA.

Crispin, C., Calton, P., Dave, M., Jonathan, W., Peat, B., Steve B., Aaron, G., Perry W., Qian Z., and Heather H., 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks. In *Proc. 7th USENIX Security Conference*, pages 63-78, San Antonio, Texas.

Crispin, C., Matt, B., Steve, B., Greg, K., Mike, F., and Jamie L., 2001. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, Washington, DC.

Jun, X., Zbigniew, K., and Ravishankar, K. I., 2003. Transparent Runtime Randomization for Security. In *Proceedings of the 22nd International Symposium on reliable Distributed Systems*, pages 260-269, Florence, Italy.

Michel, K., 2001. Vudo Malloc Tricks. *Phrack 8(57)*.

Mudge, 1997. How to Write Buffer Overflows. Published on World-Wide Web at URL *http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html*.

Nergal, 2001. Advanced return-into-lib(c) exploits (PaX case study). *Phrack 4(58)*.

PaX Team. Homepage of The PaX Team. h*ttp://pax.grsecurity.net*.

Rafal ,W., 2001. Defeating Solar Designer Non-Executable Stack Patch. *http://www.insecure.org/sploits/non-executable.stack.problems.html*.

"Solar Designer". Non-Executable User Stack. *http://www.openwall.com/linux/*.

Stephanie, F., Anil, S., and David H. A., 1997. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, pages 67-72, Los Alamitos, CA. IEEE Computer Society Press.

Tzi-cker C. and Fu-Hau H., 2001. Rad: A Compile-Time Solution to Buffer Overflow Attacks. In *21st International Conference on Distributed Computing*, page 409, Phoenix, Arizona.