Active Modification Method of Program Control Flow for Efficient Anomaly Detection

Kohei Tatara[1], Toshihiro Tabata[2], Kouichi Sakurai[3]

[1] Graduate School of Information Science and Electrical Engineering, Kyushu University,
Postal Code 812-8581
6-10-1 Hakozaki, Higashi-ku, Fukuoka, Japan
tatara@itslab.csce.kyushu-u.ac.jp
[2] Graduate School of Natural Science and Technology, Okayama University,
Postal Code 700-8530,
3-1-1 Tsushima-naka, Okayama, Japan
tabata@cs.okayama-u.ac.jp
[3] Faculty of Information Science and Electrical Engineering, Kyushu University,
Postal Code 812-8581,
6-10-1 Hakozaki, Higashi-ku, Fukuoka, Japan
sakurai@csce.kyushu-u.ac.jp

Abstract. In order to prevent the malicious use of the computers exploiting buffer overflow vulnerabilities, a corrective action by not only calling a programmer's attention but expansion of compiler or operating system is likely to be important. On the other hand, the introduction and employment of intrusion detection systems must be easy for people with the restricted knowledge of computers. In this paper, we propose an anomaly detection method by modifying actively some control flows of programs. Our method can efficiently detect anomaly program behavior and give no false positives.

1 Introduction

Buffer overflow vulnerabilities often arise from some bugs existing potentially in programs [1]. Accordingly, we can offer the following methods.

Programmers' efforts
All of the programmers should contrive and follow a common framework for secure programming. The framework consists of a set of rules, under which they can write source codes without buffer overflow vulnerabilities.

Compiler improvement for retrieval
We can adapt a compiler, and apply pattern matching or syntactic parsing techniques at the compilation of source codes, whereby we may find some vulnerable functions or parts of them.

Operating system's control
All of the application programs are certainly controlled by an operating system.

Therefore, the operating system can find anomalous behaviors of them and prevent illegal use of computers. This approach comprises some methods of excluding some vulnerability around dynamic libraries.

We can also restrain programmers from writing source codes which includes some bugs, by calling their attentions. But, as long as they are likely to overlook them, any approaches available for the users count for much. Methods using compilers may accompany the recompilation of application programs. Other methods may require the users' knowledge of vulnerabilities.

Some approaches within the pale of the operating system conversion, can suppress the influence on the operating system to minimum. Against this background, the topic of this paper focuses on them. Typically, the role of detecting the illegal use of a computer is referred to as anomaly detection system. This is classified into the follows: misuse detection system, which finds some signatures of invasive act, or anomaly detection system, which catches on the anomalous behaviors derived from the normal ones. The main advantage of anomaly detection systems is that they may detect novel intrusions, as well as various kinds of known intrusions. However, the systems can be complicated and increase the overheads. Our method can be categorized into the anomaly detection, but have following characteristics.

1. Our method can detect the buffer overflow in real time and prevent the attacker from using the computer without authorization.
2. Our method can keep the overhead for anomaly detection to a minimum. Therefore, it is easy for users to introduce our method without degrading the system performance.
3. Our method does not require the recompilation of application programs. This provides relatively easy transition from the current system. Therefore, the users need little knowledge for installation and employment.

In order for intrusion detection systems to be accepted as practical tools, a capability to detect the illegal use of computers with high accuracy and with low overhead is essential. Therefore, it is very important to satisfy the first and second requirements. Moreover, users' knowledge of operating systems and application programs is often limited. Because it is desirable that the cost for introduction and employment is as low as possible, the third requirement is expected to encourage users.

This paper is organized as follows. In chapter 2, we describe a background, against which intrusions exploiting buffer overflows vulnerabilities occur and mechanism of them. Also, we clarify novelty of our method and scope of application. Then, we explain about our method in chapter 3. Subsequently, we instantiate how to implement our method on the IA-32 Linux in chapter 4, and analyze security and efficiency in chapter 5. At last, we conclude in chapter 6.

2 Background and Motivation

2.1 Intrusions Exploiting Buffer Overflow Vulnerabilities

When running a program, a memory area, called stack, are temporarily reserved for sequential input and output data. The stack contains the First-In Last-Out data structure that pops elements in the opposite order than they were pushed. When calling a subroutine or a function, it can be used for evacuating some data in progress and a return address.

In the high level languages like C or C++, the reservation and management of local buffers relies on the programmers. They are responsible for managing or reserving local buffers appropriately. However, if careless programmers often allow an attacker to write greater length of values than they allocated previously, the attacker can happily write over the end of buffers and pollute parts of a frame pointer (fp) and a return address (ret). This is called a buffer overflow. In the process of the intrusion, the attacker intentionally causes a buffer overflow and replaces a return address or a function pointer with the address of shellcode or shared library (attack code). Figure 1 shows the appearance of stack corrupted by the buffer overflow. The attacker is able to execute arbitrary commands by modifying the control flow to run the shell program.

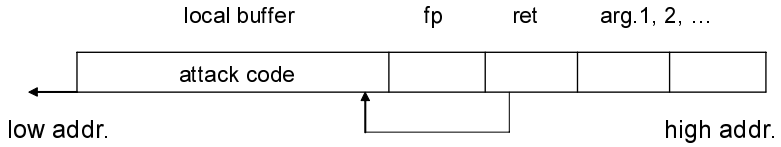


Fig. 1. Appearance of stack

2.2 Related Work and Motivation

We introduce some approaches addressing intrusion detection from the side of the operating system. They are slightly different from the approaches on the compiler's side, and do not have to recompile each application program in introducing it into the system. If they require operating system to be restarted on the occasion of introduction, all of the services on it must be halted. Therefore, in order to be provided as a kind of add-in tool, it is also more preferable that they do not have to conduct recompilation of kernel.

The Openwall [2] has functionality of not allowing processes to run any executable codes in the stack area. When an attacker stores shellcode in the stack area and substitutes an original return address with the pointer to it, this mechanism is able to catch

on it probably. Currently, some CPUs support the so-called NX (Non-Execute) bit which prohibits execution of codes that is stored in certain memory pages to prevent the intrusions exploiting buffer overflow vulnerabilities as well as Openwall. However, these are not perfect solutions, because there exists another attack which cannot be detected with non-execution of stack [3].

The StackGuard [4] is able to detect the actual occurrence of buffer overflow by checking whether a certain value, called canary, inserted in stack is changed or not. However, it cannot decide whether local variables are correct or not. When an attacker overwrites a function pointer, it may still fail to detect executions of unauthorized codes.

The PointGuard [5] prevents intrusion from occurring by encrypting function pointer in memory, even if an illegal falsification using buffer overflow is done. Because the encryption is mounted by the value exclusive-OR'ed with a random number, the overhead is small. However, PointGuard can be applied when the program is compiled, and application to execution code is not assumed. This paper concentrates on the feasibility of technique for detecting alternation of function pointer and return address under the situation in which only execution code is given.

Prasad et al. [6] proposed a method for detecting the stack overflow that uses rewriting execution code on IA-32 in detail. They touched about the difficulty of disassembling execution code. And, they clarified the coverage and made it to the measure of effectiveness. Their system can detect stack overflow by building the mechanism of RAD [7] into the execution code on Intel 32-bit Architecture (IA-32). Then, they supposed that their disassembler can catch prologue and epilogue of the function, and the existence of the area to insert the jmp instruction as a precondition [6]. We think therefore that it is appropriate that our method also puts same assumption as theirs. That is, we assume the existence of the one similar to the frame pointer or it in the function.

3 Our Proposal

3.1 Modification of Control Flow

When loading a program into the memory area, our method modifies the control flow of it. Then, it inserts an additional flow, which provides the verification process, called verification function, to check the legitimate use in the vicinity of each function call. The process of the modified function works as follows.

1. When the function is called, a stack frame is allocated for the return address. The return address in this stack frame is set to the next instruction. When the function call finishes, the process will be resumed at the return address set in the stack frame.
2. Next, the size of a pointer is taken from the value of the stack pointer. The address of the verification function is to be put in this reserved area. Then, the value of the frame pointer is put in the stack. The value of the frame pointer is

newly replaced by the value of the stack pointer. After a stack frame is allocated for the local variables, the process of the original (unmodified) function will be carried out.

3. At the end of the function, the address of the verification function is overwritten by using the frame pointer in the reserved area. Therefore, in spite that the function is finished successfully or abnormally, the control of the process will be transferred to the verification function.

3.2 Adjustment for Function Call

If there is the function call in the coverage of our method, it will be modified as follows. When calling another function, the return address that is supposed to be put in the stack is exclusive OR'ed with the random value *p*. The result is put in the stack as if it is the return address.

Next, the process of the function call is modified so that the control is transferred to the address specified by the operand of it, which is exclusive OR'ed by random value *p*. Also, the operand itself of the function call is correspondingly modified in advance. Especially, if the operand can be considered as one of the registers or a pointer to memory, we will try to find the instruction which provides the input to it. If a certain constant value is found in the instruction, it will also be exclusive OR'ed with the random value *p*. The value of *p* is chosen at every execution of the program. Here, we note that if the control is transferred to the raw address specified by the external input is called, we cannot locate such value. Such programs are not included in the coverage of our method. Because the *p* is not only secret (only known by the operating system) but also fresh (different at each execution), the operating system can only run the program normally.

3.3 Process of Verification Function

The verification function takes the random value *p* and the (exclusive OR'ed) return address as arguments. It can verify that the function was called from the valid origin. If *p* is not correct, the return address may probably be corrupted. In other word, we can decide that there exists the invasive action using the buffer overflow. In our method, the return address given as an argument will be exclusive OR'ed with the random value *p* once again, and the control will be transferred to the address pointed by the result. For this reason, only when *p* is correct, the execution will be resumed correctly.

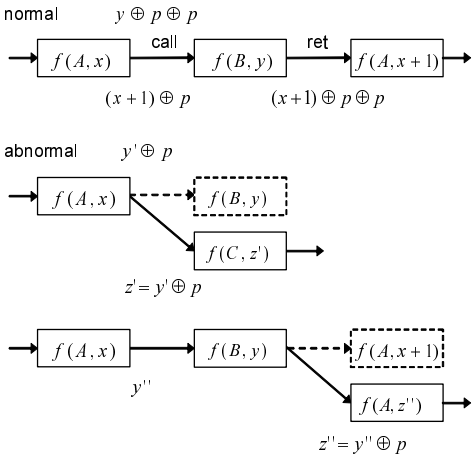


Fig. 2. State transition associated with function call

3.4 Procedure of Detecting Anomalous Behavior

We explain the procedure where our method detects the invasion act. Figure 2 shows that the function *A* calls another function *B*. The top of these flows represents normal execution, and the lower flows shows that the return address or the function pointer are illicitly replaced.

f(A,x) indicates that CPU is running *x*-th instruction in the function *A*. In our method, when *A* calls *B*, both the operand of the function call and the return address are exclusive OR'ed with the random value *p*. If its operand (e.g., function pointer) is illicitly replaced by the pointer to the address of a shellcode or a shared library (*z'*), the control is transferred to the address $z' \oplus p$. This address may point to an illegal address, and cause an error. Also, it is assumed that an attacker can replaces the return address with an address (*z"*). Regardless of buffer overflow, the verification function is certainly called. Therefore, when the process is restarted at (A, $z'' \oplus p$). If the attacker does not know the random value *p*, he cannot succeed in any intrusion.

4 Implementation

4.1 Modification of Program

In this chapter, we touch on capability of the implementation in an existing system. Specifically, we use Linux operating system (Kernel 2.4.19) working on Intel 32-bit

Architecture, and take GNU C Compiler (gcc-2.96). But, we believe that our method is applied for another platform.

```
_func1:
   pushl    %ebp
   movl     %esp, %ebp
   subl     $24, %esp
   movl     $_func2, -4(%ebp)
   movl     $LC1, 4(%esp)
   movl     $1, (%esp)
   movl     -4(%ebp), %eax
   call     *%eax
   leave
   ret
   ...
_func2:
   pushl    %ebp
   movl     %esp, %ebp
   subl     $8, %esp
   incl     8(%ebp)
   movl     12(%ebp), %eax
   movl     %eax, 4(%esp)
   movl     $LC0, (%esp)
   call     _printf
   leave
   ret
```

Fig. 3. Original code

Figure 3 shows the original (assembler) code before our method is applied. This code means that a function func2 is called in a function func1. The modified (assembler) code is shown in Figure 4. We schematically explain the behavior when running it. Note that the code modification is performed at the binary level. Here, for ease of explanation, we show the assembler codes.

1. Firstly, when func1 is called, a stack frame is reserved for the address of the verification function. At once, the frame pointer %ebp is put in the stack. the stack pointer %esp is used as a new %ebp. Then, 24-bytes are subtracted from the stack pointer to obtain a memory space for local variables.
2. The part, where there exists a call instruction (call _func2), is modified so that both the return address put in the stack and the operands of the call instruction are exclusive OR'ed with the random value *p*.
3. When the process of the original code is finished, the address of verification function is written in the reserved area by using the frame pointer %ebp. The verification function is then started.

```
_func1:
   call     _trampoline11
   nop
   movl     $_trampoline20, -4(%ebp)
   movl     $LC1, 4(%esp)
   movl     $1, (%esp)
   movl     -4(%ebp), %eax
   jmp      _trampoline12
   nop
   nop
   ...
_trampoline10:
   xorl     $0x12345678, (%esp)
   jmp      _func1
_trampoline11:
   movl     (%esp), %eax
   pushl    %ebp
   movl     %esp, %ebp
   subl     $24, %esp
   jmp      *%eax
_trampoline12:
   call     *%eax
   movl     $_ver, 4(%ebp)
   leave
   ret
_ver:
   xorl     $0x12345678, (%esp)
   ret
```

```
_func2:
   call     _trampoline21
   nop
   incl     12(%ebp)
   movl     16(%ebp), %eax
   movl     %eax, 4(%esp)
   movl     $LC0, (%esp)
   jmp      _trampoline22
   nop
   nop
   ...
_trampoline20:
   xorl     $0x12345678, (%esp)
   movl     $ _func2, %eax
   xorl     $0x12345678, %eax
   ...
   xorl     $0x12345678, %eax
   jmp      *%eax
_trampoline21:
   movl     (%esp), %eax
   pushl    %ebp
   movl     %esp, %ebp
   subl     $8, %esp
   jmp      *%eax
_trampoline22:
   call     _printf
   movl     $_ver, 4(%ebp)
   leave
   ret
   ...
```

Fig. 4. Modified code

4.2 Assuring the Consistency in the Vicinity of Modification

The most important thing to change the control flow is to minimize the effect on the original program. Also, it is important that we avoid using the variable parameters amenable to environment or external input. As shown in Figure 4, we rewrite the call instruction so that the return address is exclusive OR'ed with p (= 0x12345678) before stepping into the next function func2. However, in the verification function, the result of it is exclusive OR'ed once again. Definitely, the consistency can be preserved in the vicinity of the application of our method.

The call instruction,

```
movl    $_func2, -4(%ebp)
...
movl    -4(%ebp), %eax
call    *%eax
```

is adjusted as follows.

```
movl    $_trampoline20, -4(%ebp)
...
movl    -4(%ebp), %eax
```

```
    jmp    _trampoline12
    ...
_trampoline12:
    call   *%eax
    ...
_trampoline20:
    xorl   $0x12345678, (%esp)
    movl   $ _func2, %eax
    xorl   $0x12345678, %eax
    ...
    xorl   $0x12345678, %eax
    jmp    *%eax
    ...
```

By inserting the ``trampoline'' function, we do not have to consider any effects on former and latter instructions. Thus, as shown in Figure 3, 4, we can find that the size of func1 and func2 are kept during the modification. If the size was changed, we also had to update all the values of addresses in the latter instructions.

However, in the func2, the references to the arguments are modified as follows.

incl 8(%ebp) → 12(%ebp)

This is because we reserved the stack frame for the address of the verification function.

5 Security and Efficiency

5.1 Resistance for Intrusion

An attacker can hijack the control flow by overwriting the return address or the function pointer. But, the buffer available to the attacker is likely to be limited. Therefore, we assume that the attacker certainly uses functions or libraries supplied by the operating system. When she invades the system without using them, our method cannot work well. Namely, if she only tries to overwrite some variables, the system will condone her offence.

Modifying a return address of the function call prevents the attacker from replacing them with addresses of shellcode or shared libraries. On the other hand, the threat of replacement of function pointer cannot be eliminated in this scheme. We cannot decide whether an input to the function pointer is correct, because the pointer variable can be changed in the normal process. Our method is also resistant to this attack by modifying the operands of function call.

5.2 Possibility of Intrusion

The modification of a program code is carried out when loading it into the memory space. In order for the attacker to get the correct return address and modify it, she must guess the random value *p*. Possibility of succeeding in the attack is very small, when the attacker cannot guess it, because we assume that *p* is very large value (about 2^{32}).

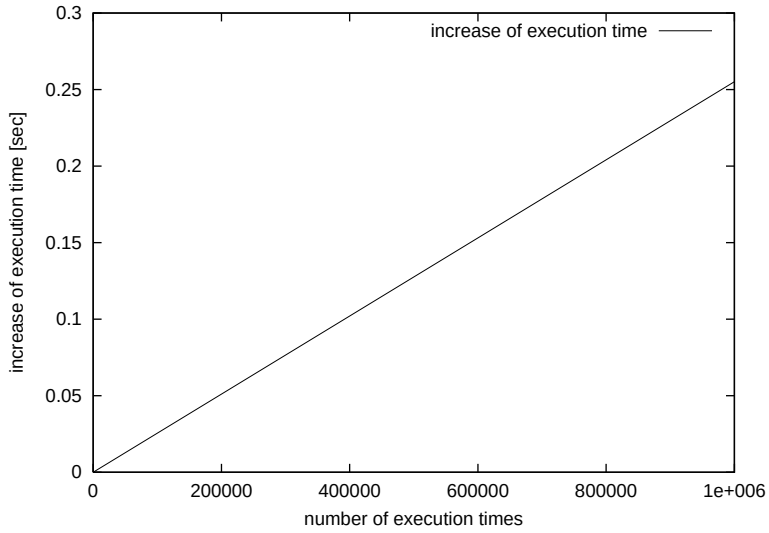


Fig. 5. Overhead per function call (x-axis: number of execution times, y-axis: increase of execution time [sec])

5.3 Efficiency

Our method is applied to each function. We measured how much overhead is required to adapt our modification comparing the basic performance. Specifically, we saw the overhead per function call provided by executing the original code and modified code in our method (Figure 3, 4). Our machine has Pentium 4 1GHz, 512MB RAM. The result of fifteen sets of one million executions shows that the (average) increase of execution time is 0.255137 seconds, and the relation between the increase and the number of execution times in Figure 5. In order to be accepted as practical art, our method should be applied to the application programs widely used, and be evaluated in spite that this is negligible small value.

6 Conclusion

The case of abuse of computers using buffer overflow are continually reported and considered as serious problem. Currently, some CPUs support the so-called NX bit (Non-Execute) which prohibits the execution of code that is stored in certain memory pages to prevent the intrusion exploiting buffer overflow vulnerability. However, this is not a perfect solution, because there is another attack which cannot be detected with non-execution of stack.

In this paper, we proposed an anomaly detecting method by modifying the control flow of the program. Our method has advantage of no false positives and reducing the overhead. It is also expected that it only takes the costs of the introduction and employment, and encourages users to install it. Future work is further evaluation of anomaly detection system.

References

[1] Beyond-Security's SecuriTeam.com, "Writing Buffer Overflow Exploits - a Tutorial for Beginners," http://www.securiteam.com/securityreviews/5OP0B006UQ.html (accessed 2003-09-05).

[2] Openwall Project, "Linux kernel patch from the Openwall project," http://www.openwall.com/linux/ (accessed 2004-01-20).

[3] Linus Torvalds, http://old.lwn.net/1998/0806/a/linus-noexec.html (accessed 2004-02-13)

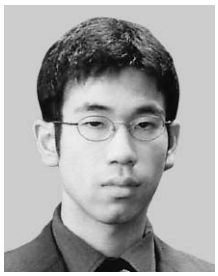
[4] P. Wagle and C. Cowan, "StackGuard: SimpleStack Smash Protection for GCC," In Proceedings of the GCC Developers Summit, pp. 243–255, May 2003.

[5] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Point-Guard: protecting pointers from buffer overflow vulnerabilities," In Proceedings of the 12th USENIX Security Symposium, Aug. 2003.

[6] M. Prasad and T. Chiueh, "A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacksk," In Proceedings of Usenix Annual Technical Conference, Jun. 2003.

[7] T. Chiueh and F. Hsu, RAD, "A compile time solution for buffer overflow attacks," In Proceedings of 21st IEEE International Conference on Distributed Computing Systems (ICDCS), Apr. 2001.

Biography



▲ Name: Kohei Tatara

Address: 6-10-1 Hakozaki, Higashi-ku, Fukuoka, Japan

Education & Work experience: eceived the B.E. degree in 2004 and the M.E. degree in 2006 all from the Kyushu University, Fukuoka, Japan. He is working for the Ph.D. in engineering at Kyushu University.

Tel: +81-92-642-3867

E-mail: tatara@itslab.csce.kyushu-u.ac.jp

Other information: His interests include operating system and intrusion detection. His is a member of the Information Processing Society of Japan (IPSJ).

▲ Name: Toshihiro Tabata

Address: 3-1-1 Tsushima-naka, Okayama, Japan

Education & Work experience: received the B.E. degree in 1998, the M.E. degree in 2000, and Ph.D. degree in engineering in 2002, all from Kyushu University, Fukuoka, Japan. He had been a research associate of Information Science and Electrical Engineering at Kyushu University since 2002. He has been an associate professor of Graduate School of Natural Science and Technology, Okayama University since 2005.

Tel: +81-86-251-8188

E-mail: tabata@cs.okayama-u.ac.jp

Other information: His interests include operating system and computer security. He is a member of the Information Processing Society of Japan (IPSJ), IEICE, ACM and USENIX.

▲ Name: Kouichi Sakurai

Address: 6-10-1 Hakozaki, Higashi-ku, Fukuoka, Japan

Education & Work experience: received the B.S. degree in mathematics from Faculty of Science, Kyushu University and the M.S. degree in applied science from Faculty of Engineering, Kyushu University in 1986 and 1988, respectively. He had been engaged in the research and development on cryptography and information security at Computer & Information Systems Laboratory at Mitsubishi Electric Corporation from 1988 to 1994. He received the Dr. degree in engineering from Faculty of Engineering, Kyushu University in 1993. Since 1994 he has been working for Department of Computer Science of Kyushu University as an associate professor, and now he is a full professor from 2002.

Tel: +81-92-642-4051

E-mail: sakurai@csce.kyushu-u.ac.jp

Other information: His current research interests are in cryptography and information security. Dr. Sakurai is a member of the Information Processing Society of Japan, the Mathematical Society of Japan, ACM and the International Association for Cryptologic Research.