Secure Call and Return Instructions for Mimicry Attack Detection

Yuuki Tominaga*, Takehiro Kashiyama*, Eiji Takimoto*, Hiroaki Kuwabara*,
Koichi Mouri*, Shoichi Saito†, Tetsutaro Uehara‡ and Yoshitoshi Kunieda*,
*Ritsumeikan University, Shiga, Japan
†Nagoya Institute of Technology, Nagoya, Japan
‡The Research Institute of Information Security, Wakayama, Japan

I. Introduction

Some kinds of Attack using program vulnerabilities (e.g. buffer overflow vulnerability) change the control flow of program to be contrary to the design of developers. If attacked program runs in privilege level, secret information leakage or falsification of data may be occurred. Many methods have been proposed to detect the occurrence of buffer overflow using host-based intrusion detection systems (IDSs) [1]. IDSs detect malicious attacks by checking whether the behavior of running program follows pre-defined correct ones. However, most IDSs cannot detect malicious attacks when attacked program behaves like unattacked one. Wagner et al. found that IDSs can be avoided if the attack code imitates the original program [2]. Such kind of attack is called as "Mimicry Attack." Kruegel et al. show how to automate Mimicry Attacks using static binary analysis [3]. This means it is easy to execute Mimicry Attacks.

In this paper, to detect Mimicry Attacks, we newly propose a secure extension of call and return instructions of well-known Intel x86 architecture and including any compatible processors that has the stack frame. These extended instructions verify the control data (the value of return address and frame pointer) in the call stack. Extended call instruction saves the control data to hidden area and extended return instruction checks the control data in call stack and hidden area. This mechanism is able to detect rewritten control data and protect the running programs against Mimicry Attacks.

II. Mimicry Attack

Mimicry Attacks rewrite all stack frames by using buffer overflow to show another call stack which is possible to appear while the program is running. Since rewritten stack frames show the correct behavior of program, IDSs using static analysis cannot detect Mimicry Attacks.

Fig.1 shows an example of Mimicry Attacks. In this example, `Function1` calls `Function2` and `Function4`, `Function2` calls `Function3`, `Function4` calls `Function5` and `Function5` calls `Function6`. We assume `Function3` has a buffer overflow vulnerability. In Fig.1, (1) shows the stack frames while `Function3` is running. Mimicry Attacks use vulnerability of `Function3` to rewrite stack frames that show `Function6` is running like Fig.1 (2). In this case, return instruction in `Function3` returns program control to `Function5` not to `Function2`.

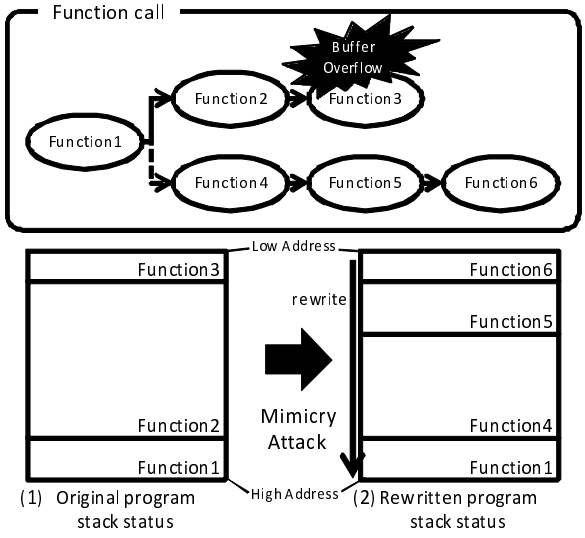

Fig. 1. An Example of Mimicry Attack.

This illegal behavior is difficult to detect since there is no record showing the control moves from `Function3` to `Function5` after return executed. Traditional IDSs check the order of function call by using stack frames whenever any system calls are called. So, it is impossible to detect the occurrence of Mimicry Attacks in the functions which use no system calls.

III. Extended call and return instructions

In this section, we describe a secure extension of call and return instructions as a method for Mimicry Attack detection. Mimicry Attacks rewrite all control data on the call stack to reflect that program reaches any specific functions through the correct control flow. This makes possible to move the program control to any functions as a legal behavior. Since Mimicry Attacks need to rewrite control data, it is able to prevent Mimicry Attacks by checking whether some control data are rewritten when return instruction is executed.

The overview of our proposed method showed in Fig. 2. This detection method is composed of a saving process and a checking process of the control data. These processes are called whenever function is called or returned. For implement

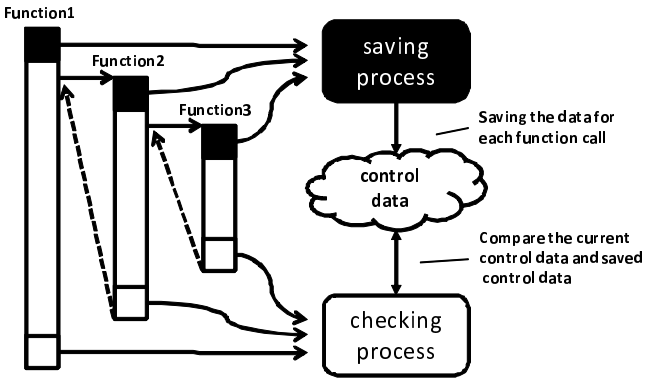

Fig. 2. Overview of our proposed method.

these processes, the extension of call and return instruction are utilized in our proposed system.

A. Extended call instruction

Extended call instruction saves the control data on the call stack to some hidden area after traditional function calling process. We suppose this hidden area is prepared on the processor chip and should not be access any instructions except for extended call and return. Saving process uses hidden area as a stack. We call this stack "protection stack."

There are two ways following to save the control data to the protection stack. (1) The control data that is stacked on the lowest address is saved to the protection stack whenever extended call instruction is executed. (2) The hashing value of all control data on the stack is saved to protection stack whenever extended call instruction is executed. The detection accuracy and the using memory are relationship as trade-off. In case of (1), the detection accuracy is the highest, however the amount of using memory is big. In case of (2), the amount of using memory is smaller than way of (1), however this way has a problem that overlook the attacks, when the function call order is changed to another one that has same hashing value.

B. Extended return instruction

Extended return instruction checks whether some control data are rewritten by comparing the data on real call stack and protection stack before traditional returning process. This checking process can detect Mimicry Attacks have been prepared since Mimicry Attack need to rewrite the control data before firing. Then, checking process kills the attacked program by interruption before prepared attack fires.

IV. Simulation and Evaluation

A. Simulation

We implement our method as software simulation for proof of concept. The saving and checking processes are implemented as system calls. The hidden area for these processes is allocated in the kernel space. To simulate extended call instruction, some codes to invoke saving process are inserted into the head of each function. Similarly, some codes to invoke checking process are inserted into the last of each function to simulate extended return instruction. These code insertion are performed by using GCC option "-finstrument-functions." We use sysenter instruction to invoke system calls of saving and checking processes.

TABLE I
Evaluation environment.

CPU	Intel Core Duo（1.86 GHz）
Memory	1 GB
OS	Fedora 15
Kernel	Linux Kernel 2.6.38

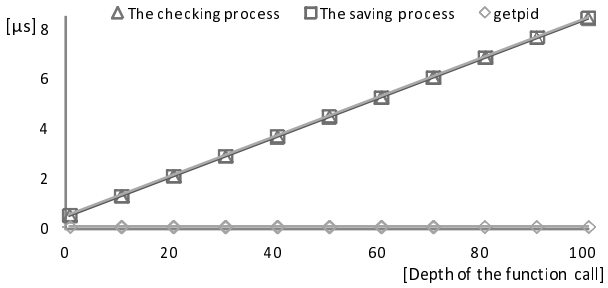

Fig. 3. Saving and checking time.

B. Sample Mimicry Attacks

We prepare sample target programs which have buffer overflow vulnerability or format string vulnerability [4]. These vulnerabilities can be used for Mimicry Attacks. We attack these programs by Mimicry Attacks like Fig.1. First, some call stack showing specific function is invoked is dumped. Next, the call stack is rewritten to mimic dumped stack by using program vulnerabilities. Without our method, attacked programs showed malicious behavior based on the rewritten stack. On the other hand, with our method, attacked programs are killed when Mimicry Attacks are detected.

C. Overhead of our approach

We expound the overhead toward the control data of evacuating and inspection process, and gzip and httpd, which are practical applications. Table I shows the evaluation environment.

Overhead of saving and checking processes: We measured the processing overhead in the saving process and the checking process. The saving process and the checking process needs to call system call for processing each process. To compare the processing time without the system call processing time, we also measured getpid system call. The results are shown in Fig.3. The processing of getpid system call is the smallest in common system calls. Most of the processing of getpid system call is the mode change as moving from user mode to kernel mode or from kernel mode to user mode. The main processing of our proposed method is to extract the control data between the highest address from the lowest address in

TABLE II
OVERHEAD OF GZIP PROCESS.

		Run time (sec)	Overhead
1.	Normal	5.477	–
2.	Proposed	5.619	2.74%

TABLE III
OVERHEAD OF HTTPD PROCESS.

	Normal	Proposed	
	Time per request(ms)	Time per request(ms)	Overhead
4 KB file	0.198	1.567	690.41%

stack. As a function call is repeated, the processing overhead of extracting the control data will be big. It is shown in Fig.3. The actual overhead of the program depends on the frequency and depth of the function call.

Overhead with gzip: We apply our method to gzip application (version 1.2.4), and confirm that the gzip application works correctly without false positive. In addition, we measured the running time of gzip application with decompress of Linux kernel source code. In Table II, we show the average time of 1000 times measure. The total of function call was 51,843 times when running gzip program. The average of function call's depth was 8.421, and then it was cleared that there are no recursive functions in gzip program. Running time of application was 5.477 sec without our proposed system, 5.619 sec with our proposed system. The percentage of overhead by them was increased to 2.59%. In case of decompress process the percentage of program was small, because the time of function process was more than overhead by them.

Overhead with httpd: We apply our method to httpd application: (Apache HTTP Server 2.2.21), and confirm whether the operation is correct, and measured the value of the overhead. Because httpd is restless program as daemon, it is difficult for us to measure the processing time from the beginning to the end. So, using Apache Bench [5], we measured the time required for the processing of one request and overhead. The result is shown in table III. We measured with the machine run by httpd and Apache Bench. The server machine is connected to measuring machine with a straight cable. We measured the time of processing of one request that 10,000 times issue 4 KB file which was accessed by 100 lines at the same time. The running time of application is 0.198 msec without our proposed system, 1.567 msec with our proposed system. In case of httpd application, the running of one request was small (0.198 msec). Therefore, the percentage of overhead became to about 690%. It was cleared that httpd application's overhead (1.369 msec) was small compared with gzip application's overhead (142 msec).

V. DISCUSSION

In the software simulation, when the function is called, saving process is done, and when the function is returned to calling function, checking process is done. In the saving process of control data, the control data which is not influenced by the buffer overflow is stored in protection stack in kernel memory. The checking process is judged whether control data is already rewritten is not, by comparing current control data in the call stack with control data that is saved. From the result of software simulation, we confirmed that Mimicry Attack is detected certainly by our proposed system. It show that the inspection of our proposed system is extremely precise than another IDSs. Moreover, our proposed system can inspect the rewriting of control data, and then defend against code injection attack and return-into-libc attack.

When applying the proposed system to practical application, the overhead return 2.74% to 690%. Sendmail applying by Wagner's method [6] indicated overhead more than one hour. In case of implement the proposed system on the hardware, the time will be short to call the system calls, so we think that it can cut down the overhead very much. In addition, it is possible to add security easily without changing the software, so we think that there is a convenience.

VI. CONCLUSIONS

This paper proposes new detection method of Mimicry Attacks which are undetectable by most existing IDSs. Our proposed method consists of the saving which saves the reutrn address and the checking process which inspects whether the return address is not rewritten. The saving process is called after traditional function calling process, the checking process is called before traditional returning process. By implement the simulation of our mechanism, we verified that it can detect certainly Mimicry Attack. The overhead of execution time was probed 2.74% increasing for gzip and 690% for httpd.

As future works, this mechanism should be implemented into usual call and return instruction. As the saving process in the calling sequence of this mechanism is only a duplication of the existing call instruction, the amount of newly needed hardware will be small and it can be assumed that the total execution time overhead described above can be reduced.

REFERENCES

[1] S. Forrest, S. Hofmeyr, S. Somayaji, and T. Longstaff, "A sense of self for unix processes," in *Proceedings of 1996 IEEE Symposium on Security and Privacy*, 1996, pp. 120–128.

[2] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM conference on Computer and communications security*, 2002, pp. 255–264.

[3] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *Proceedings of the 14th conference on USENIX Security Symposium*, 2005, pp. 11–26.

[4] U. Shankar, K. Talwar, J. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *Proceedings of the 10th conference on USENIX Security Symposium*, 2001, pp. 16–31.

[5] ApacheBench, "A complete benchmarking and regression testing suite," http://httpd.apache.org/.

[6] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001, pp. 156–168.