I.J. Information Technology and Computer Science, **2009, 1, 1-7**
Published Online October 2009 in MECS (http://www.mecs-press.org/)

A System Call Randomization Based Method for Countering Code-Injection Attacks

Zhaohui Liang
School of information Renmin University of China
Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China)
lzh@ruc.edu.cn

Bin Liang
School of information Renmin University of China
Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China)
liangb@ruc.edu.cn

Lupin Li
School of information Renmin University of China
li_lu_ping@sina.com

***Abstract*—Code-injection attacks pose serious threat to today's Internet. The existing code-injection attack defense methods have some deficiencies on performance overhead and effectiveness. To this end, we propose a method that uses system called randomization to counter code injection attacks based on instruction set randomization idea. System calls must be used when an injected code would perform its actions. By creating randomized system calls of the target process, an attacker who does not know the key to the randomization algorithm will inject code that isn't randomized like as the target process and is invalid for the corresponding de-randomized module. The injected code would fail to execute without calling system calls correctly. Moreover, with extended complier, our method creates source code randomization during its compiling and implements binary executable files randomization by feature matching. Our experiments on built prototype show that our method can effectively counter variety code injection attacks with low-overhead.**

***Index Terms*—information security, code-injection attack, system call randomization**

I. Introduction

Code injection attacks are to exploit software vulnerabilities and inject malicious code into a 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) [1-5]. 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 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.

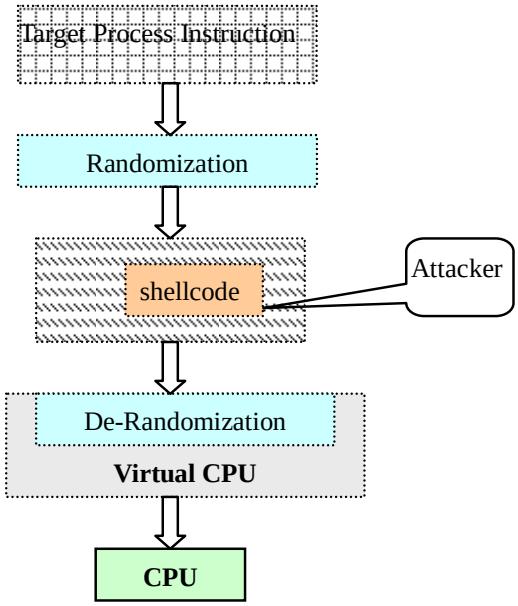

Figure 1. Instruction Set Randomization

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. 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

Manuscript received Marchl 14, 2009; revised June 22, 2009; accepted September 27, 2009.

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. Principle

The majority of injected code is machine instruction, so we focus on machine instruction code injected 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 are randomized. The original system call number Xo is overwritten with a new value Xn, calculated by the equation (1):

$$Xn = f(Xo, r) \quad (1)$$

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):

$$Xo = f^{-1}(Xn, r) \quad (2)$$

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..

Attackers 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.

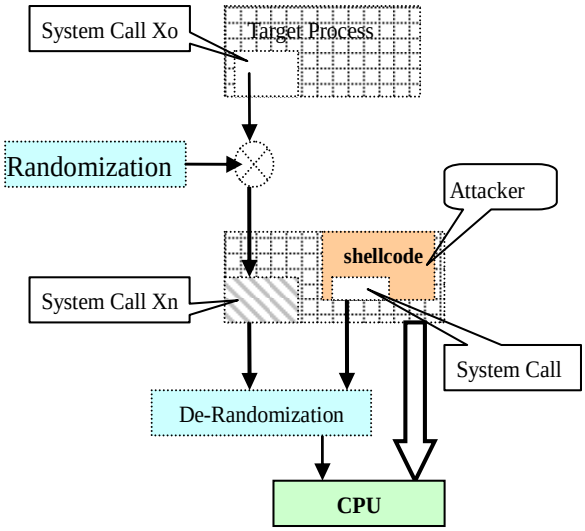

Figure 2. System Call Randomization and De-randomization

III. Implementation

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

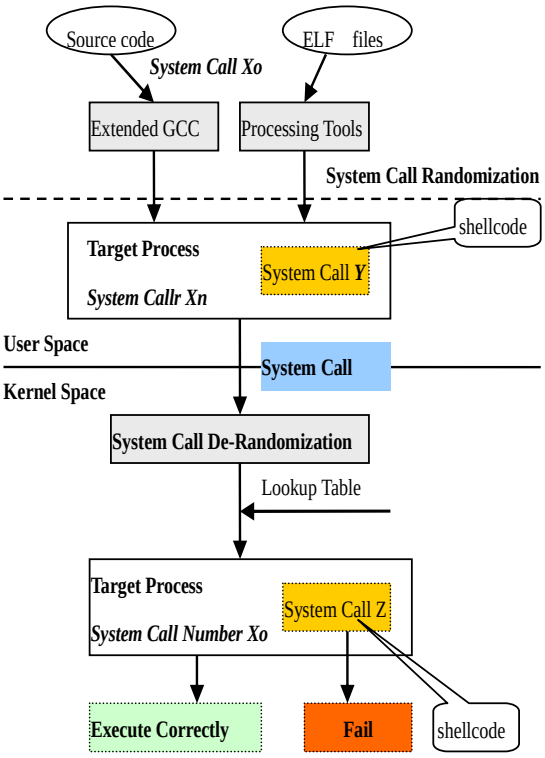

Figure 3. The Prototype of Our System

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 without source code is also implemented. As a consequence, our system can provide full protection against code injection attackers.

1) Extended GCC

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. The instruction node "*int $0x80*" contains information about system call requests, shown in Figure 4.

```
6729 (insn 163 162 164 (parallel [
6730             (set (reg/v:SI 83 [ resultvar ])
6731                 (asm_operands/v:SI ("movl %1, %%eax
6732  int $0x80
6733  ") ("=a") 0 [
6734                         (const_int 106 [0x6a])
6735                         (reg/v/f:SI 60 [ name ])
6736                         (reg/f:SI 84)
6737                     ]
6738                      [
6739                         (asm_input:SI ("i"))
6740                         (asm_input:SI ("b"))
6741                         (asm_input:SI ("c"))
6742                     ] ("../sysdeps/unix/sysv/linux/i386/xstat.c") 85))
6743             (clobber (reg:QI 19 dirflag))
6744             (clobber (reg:QI 18 fpsr))
6745             (clobber (reg:QI 17 flags))
6746             (clobber (mem:BLK (scratch) [0 A8]))
6747         ]) -1 (nil)
```

Figure 4. System call instruction node

Through feature matching, the extended GCC can identify these instruction nodes. 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, in the *6734* row of Figure 4, the system call number is *106* from the sentence "*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 from the sentence "*reg/f:SI 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 then perform the randomization. The transformed system call number will be used to construct a *const_int* type *rtx*. This *rtx* replaces the old *rtx* constructed by the original system call number. An example is shown in Figure 5. In Figure 5 the old *rtx* is *u.fld[5].rtx–> u.fld[1].rtx*.

```
printf("new_id:%d--old:%d\n",new_id,INTVAL(SET_SRC(PATTERN(Pinsn))));
Pinsn->u.fld[5].rtx->u.fld[1].rtx=gen_rtx(CONST_INT,VOIDmode,new_id);/
place*/
printf("success!\n");
print_rtl_single (fp1, insn);
if(GET_CODE(XEXP(Pinsn,5))==EXPR_LIST)/*change the other place*/
```

Figure 5. System call number transformation

To the latter, GCC will forward searches the two-way linked list from the current node to find the register which contains the system call number , search the instruction node which performs the last assignment operation to this register, and read the system call number, then follow the same steps as the former.

2) Extended GLIBC

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 PESUDO, another is 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 *DO_CALL*. In the file `glibc-2.3.6\sysdeps\unix\sysv\linux\i386\sysdep.h`, the definition of *DO_CALL* is modified as shown in Figure 6.

```
#undef  DO_CALL
#define DO_CALL(syscall_name, args)
    PUSHARGS_##args
    DOARGS_##args
    pushl $SYS_ify (syscall_name)
    call change
    addl $4,%esp
    ENTER_KERNEL
    POPARGS_##args
```

Figure 6. Modification of *DO-CALL*

In Figure 6, the original instruction "*movl*" is replaced by the other three assembly instructions. The first one is "*pushl $SYS_ify(syscall_name)*", the system call number is transferred as a parameter to the user-defined function. The second one is "*call change*". The function "*change"* is defined by user and performs the randomization. The third one is "*addl $4,%esp*" , it is to maintain the balance of stack when the function call returns.

To the latter, the core work is to modify the definition of *INLINE_CALL*. In the file `libc-2.3.6\sysdeps\unix\sysv\linux\i386\sysdep.h` the definition of *INLINE_CALL* is modified as shown in Figure 7.


```
# define INTERNAL_SYSCALL(name, err, nr, args...) \
  ({                                              \
    register unsigned int resultvar;              \
    int newid;                                    \
    newid=change(__NR_##name);                    \
    EXTRAVAR_##nr                                 \
    asm volatile (                                \
    LOADARGS_##nr                                 \
    "movl %1, %%eax\n\t"                          \
    "int $0x80\n\t"                               \
    RESTOREARGS_##nr                              \
    : "=a" (resultvar)
    : "g" (newid) ASMFMT_##nr(args) : "memory", "cc");
    (int) resultvar; })
```

Figure 7. Modification of *INLINE_CALL*

In Figure 7, the variable "*newid*" is added to receive the returned value of the randomization function "*change*". The input restriction of inline assembly is modified, the immediate number *__NR_##name* can be replaced by the variable "*newid*".

3) Binary executable files

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, so the first step is to find the system call instructions in ELF files.

We have statistics for system call number transmission instruction shown in table I

The statistics results illuminate that more than 98 percent system call number can be recognized by the instruction "*mov x,eax*". The system call request can be identified by the instruction "*int 0x80*"

TABLE I
STATISTICS FOR SYSTEM CALL NUMBER TRANSMISSION INSTRUCTION

Transmission system call number	*Number of system calls* in Binary Executable File		
	File A (53)	File B (58)	File C (67)
Using "*mov x,eax*"	52	57	67
Not using "*mov x,eax*"	1	1	0
Ratio of using "*mov x,eax*" to all	98.11%	98.28%	100%

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.

So the system call number can be obtained by feature matching and randomized by user-defined algorithm.

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. If the current process is the target process which the user wants to protect, the kernel de-randomizes the system call number using the method provided by the user, and then invokes the corresponding system call handler. Otherwise, the original system call handler will be invoked. Only the system call number in the target process will be de-randomized.

During the de-randomization, the parameters transmission can be handled by modification of function pointers in general,. But some extraordinary system calls need be processed specially, such as *sys_clone(struct pt_regs regs)*. The inline assembly language is introduced to transfer parameters as follows.

orig_sys[0]=orig_syscall[syscall_num];

asm("movl %ebp,%esp\n\t";

"popl %ebp\n\t";

*"jmp *(orig_sys)");*

The first sentence is to obtain the address of the system call table; the second sentence and the third sentence are to recover the values of EBP and ESP register, so that the stack pointer goes back to the state before loading de-randomization module. Final, the corresponding system call function is called by the instruction "*jmp*". 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.

Given randomization algorithm f and randomization factor r, we proposed an algorithm to obtain reserve function f^{1}, using equation (1) and equation (2) the system can calculate the randomized system call numbers for all original system call numbers. 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. We also proposed an algorithm to keep the randomized system call numbers within the range, because most operating systems can only support limited system call numbers.

We employed the writing-reading trigger mechanism of *procfs (process file system)*. That is, when there is a writing operation on a *procfs* file, a reading operation is triggered. By this means, most of the de-randomization work can be preprocessed.

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 II

TABLE II
OVERHEAD ON SINGLE SYSTEM CALL

	add	*getpid*	*sethostname*	*open*
Time without CIAS (μ sec)	0.00255	0.470	0.516	1.879
Time with CIAS (μ sec)	0.00255	0.532	0.579	1.984
Overhead	0.00%	13.19%	12.21%	5.59 %

Table II 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 III.

TABLE III
OVERHEAD ON SINGLE SYSTEM CALL

	tar	*gzip*	*cp*	*gcc*
Time without CIAS (sec)	1.61	8.57	0.58	11.48
Time with CIAS (sec)	1.67	8.60	0.62	11.51
Overhead	3.73%	0.35%	6.90%	0.26%

By comparison with Table II, Table III 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. It may explains that loading CIAS has a little of influence on the performance of applications.

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

Table IV 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

There are two main randomization techniques proposed: one is ISR [1-5], another is Address Space Layout Randomization (ASLR) [6-9]. ISR creates a randomized instruction set for each process so that instructions in shellcode fail to execute correctly even though attackers have already hijacked the control flow of the vulnerable process. ASLR, instead, randomizes the memory address layout of a running process (including library, heap, stack, and relative distances between data and code) so that it is hard for attackers to locate injected shellcode or existing program code, preventing attackers from hijacking the control flow. Though ISR is a powerful technique, it incurs more performance penalty, because it requires the introduction of an emulator and the binary transformation of applications. ASLR 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 [10] pointed out that some attack techniques such as Heap Spraying could bypass ASLR. RandSys [11] 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 [12,13] encrypts control information in a stack by XOR-ing it with a random number. 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 [14], the cost of StackGuard is up to 30%. Monica Chew [15] proposed several methods of mitigating buffer overflows by introducing randomness into the implementation of system software. One of their methods changes the mapping between system call IDs and system call handlers by mixing up the system call table using random numbers. This is achieved by recompilation of the kernel and binary rewriting of applications to fit them to the new kernel. In their method, one mapping is shared by all processes and does not change except when the kernel is recompiled. Yoshihiro Oyama [15] 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.

TABLE IV
UNIXBENCH RESULTS

Unload CIAS			
TEST	BASELINE	RESULT	INDEX
Arithmetic Test (type = double)	29820.0	529258.9	177.5
Dhrystone 2 using register variables	116700.0	3472143.9	297.5
Execl Throughput	43.0	3241.4	753.8
File Copy 1024 bufsize 2000 maxblocks	3960.0	207819.0	524.8
File Copy 256 bufsize 500 maxblocks	1655.0	82188.0	496.6
File Copy 4096 bufsize 8000 maxblocks	5800.0	278457.0	480.1
Pipe Throughput	12440.0	668401.1	537.3
Process Creation	126.0	10498.9	833.2
Shell Scripts (8 concurrent)	6.0	101.0	168.3
System Call Overhead	15000.0	384542.1	256.4
FINAL SCORE			396.6
Loaded CIAS			
TEST	BASELINE	RESULT	INDEX
Arithmetic Test (type = double)	29820.0	529350.7	177.5
Dhrystone 2 using register variables	116700.0	3472105.8	297.5
Execl Throughput	43.0	3270.3	760.5
File Copy 1024 bufsize 2000 maxblocks	3960.0	206669.0	521.9
File Copy 256 bufsize 500 maxblocks	1655.0	82612.0	499.2
File Copy 4096 bufsize 8000 maxblocks	5800.0	277449.0	478.4
Pipe Throughput	12440.0	650828.4	523.2
Process Creation	126.0	10249.1	813.4
Shell Scripts (8 concurrent)	6.0	101.0	168.3
System Call Overhead	15000.0	374340.5	249.6
FINAL SCORE			393.7

VI. Conclusion

We described our randomization scheme on the level of OS kernel to counter code injection attacks. We randomize only the system call numbers rather than the entire instruction set, hence effectively solve the performance problem of ISR. We have also developed some techniques to enhance the system performance. Firstly, the preprocessing is introduced. A majority of tasks of the randomization and de-randomization were pre-processed when the system started to run. The table was built to store the mapping between the original system call numbers and the randomized ones. The de-randomization can be accomplished quickly by looking-up the table. So that the performance cost of our system is very low. Secondly, the target program is configurable by users. The security of a system is usually determined by the vulnerable area which should be protected firstly. In our system, users can configure the target program according to their own demand. It is more flexible during deploying the system and reduces the system overhead greatly. Thirdly, a complete protecting tool set is achieved, which can provide full protection against code injection attacks. We implement source code randomization by extended GCC and GLIBC and binary executable files randomization by matching the system call instructions in the ELF files. Finally, a dynamic randomization policy is employed. Traditional randomization policy is static, such as random numbers and encryption algorithms (AES, XOR) which are restricted in security and robustness. The dynamic randomization policy isn't dependent on the random numbers and the encryption algorithms and is configurable by users. So that it is more secure from attack.

The experiments show that our prototype system can effectively thwart a great deal of code injection attacks with low overhead.

Acknowledgment

The authors wish to thank Wenchang Shi, Wei Chen, Qingqing Kang, and Yingqin Gu. This work was supported by the National Natural Science Foundation of China under Grant No.60703102 and No.60873213; the Beijing Science Foundation under Grant No.4082018; the National 863 High-tech Program of China under Grant No.2007AA01Z414.

Reference

[1] Elena Gabriela Barrantes, David H. Ackley,Stephanie Forrest, Trek S. Palmer, Darko Stefanovi′ c, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In Proceedings of the 10th ACM Conference on Computer and Communications

Security (CCS 2003), P. 281–289, Washington DC, Oct. 2003.
[2] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In Proceedings of the 10th ACM Conference on Computerand Communications Security (CCS 2003), P.272–280, Washington DC, Oct. 2003.
[3] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Darko Stefanovic, and Dino Dai Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," ACM Transactions on Information and System Security, 2005.
[4] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis, "Building a Reactive Immune System for Software Services," In Proceedings of the USENIX Annual Technical Conference, P.149 - 161, Anaheim, CA, April 2005.
[5] Noritaka Osawa. A Smart Virtual Machine for Heterogeneous Distributed Environments: PivotVM. Transactions on Information Processing Society of Japan, 40(6):2543–2552, June 1999.
[6] PaX Team: PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.
[7] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Ef_cient Approach to Combat a Broad Range of Memory Error Exploits. Proceedings of the 12th USENIX Security Symposium, Washington, DC, USA, August 2003.
[8] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Ef_cient Techniques for Comprehensive Protection from Memory Error Exploits. Proceedings of the 14th USENIX Security Symposium 2005 , Baltimore, August 2005.
[9] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In Proc. of 22nd Symposium on Reliable and Distributed Systems (SRDS) , Florence, Italy, October 2003.
[10] Michal Bucko, Exploitation for Phun and Profit, HACKPL Security Lab
[11] Xuxian Jiang, Helen J.Wang et al, RandSys:Thwarting Code Injection Attacks with System Service Interface Randomization, Reliable Distributed Systems, page 209-218, 2007
[12] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In Proceedings of the 7th USENIX Security Symposium, pages 63–78, San Antonio, January 1998.
[13] Perry Wagle and Crispin Cowan. StackGuard:Simple Stack Smash Protection for GCC. In Proceedings of the GCC Developers Summit, pages 243 – 255, Ottawa, Canada, May 2003.
[14] Perry Wagle and Crispin Cowan. StackGuard: Simple Stack Smash Protection for GCC. In Proceedings of the GCC Developers Summit, P.243–255, Ottawa, Canada, May 2003.
[15] Monica Chew and Dawn Song. Mitigating Buffer Overflows by Operating System Randomization.Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.
[16] Yoshihiro Oyama, Akinori Yonezawa, Prevention of Code-Injection Attacks by Encrypting System Call Arguments, Technical Report TR06-01,The Univ. of Tokyo,2006

Zhaohui Liang was born in P.R. China in 1968, earned B.S. degree in the field of communication engineering in 1989 and M.S. degree in the field of pattern recognition and artificial intelligence in 1992 from Huazhong University of Science and Technology, Wuhan city, P.R.China, earned Ph.D. in the field of network communication in 2005 from Beijing University of Posts and Telecommunications, Beijing , P.R. China.

She is now a LECTURER at the school of information, Renmin University of China (RUC). Before joining RUC, she was a research engineer at Institute of Automation, Chinese Academy of Science for 6 years. Recently she has published 14 research papers and 2 books in the area of Computer Science and Communication.

At the present time, she takes part in many research projects in the area of information security supported by National Natural Science Foundation of China under Grant No.60703102 and No.60873213; the Beijing Science Foundation under Grant No.4082018; the National 863 High-tech Program of China under Grant No.2007AA01Z414 respectively. Her current research interests include information security and wireless communication.

Dr. Zhaohui Liang is a member of China Computer Federation.

Bin Liang was born in P.R. China in 1973, earned B.S. degree in the field of computational mathematic and application software, earned Ph.D. in the field of computer science in 2004 from Institute of Software, Chinese Academy of Science (ISCAS). He is now an ASSOCIATE PROFESSOR at the school of information, Renmin University of China (RUC). Before joining RUC in 2006, he did postdoctoral research in the department of computer science at Tsinghua University, aim at host security and software security analysis. His current research interests include information security and static analysis.

Luping Li was born in P.R.China in 1986, earned B.S. degree in the field of computer application technology in 2009 from Renmin University of China.

