**CENTRAL PART OF THE PAPER**

**STID(59290)**

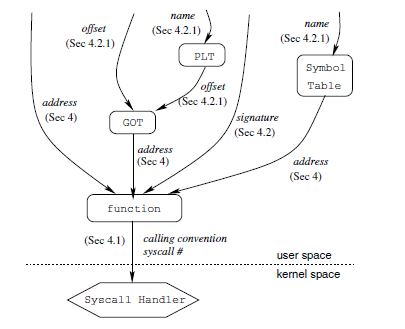
**The system call mechanism**

The kernel os basically perform all the operations and the request to kernel is given by the user level. Which use the system call interface for the purpose. This interface varies from system to system. We are describing here the mechanism of system call implemented in linux .

To invoke a system call, a user level process places the arguments to the system call in hardware registers %ebx, %ecx, %edx, %edi, and %esi (additional arguments, if any, are passed on the runtime stack); loads the system call number into register %eax; and then traps into the kernel using a special interrupt instruction, ‘int 0x80.’ The kernel then uses the system call number to branch to the appropriate code to service that system call. One effect of executing the int instruction is to push the value of the program counter. since this is done by the hardware immediately before control passes to the kernel, this value cannot be spoofed by attack code, and therefore serves as a reliable indicator of the location from which the system call was invoked.

**The attack model**

The main focus of the paper is on remote code injection attack.let suppose that the attacker contains the code for the victim application which he wanted to attack so for prevention from this attack we randomize our binary for all single specific randomize instances for all host based system. Now after the inforecemtnof this logic it is very hard for the attacker to find out the system instructions and sequences.



This figure is basically illustrating our attack model. Rectangles represent functions and data structures; arrows represent information necessary to access them.for example if the attacker wanted to attack he simply would need the system call calling convention and proper system call number by knowing these two thing attacker can attack by synthezing the proper code more or less if the attackers knows the function code he can also perform proper system call Our methods of making the necessary information unavailable to the attacker are described in subsequent sections, as indicated by the labels on the arrows.

**Constructing Interrupt Address Tables:**

The linux kernel to incorporate IAT information is modified to data structure of kernel . now first we check after the execution of executable file that if the file contains an IAT or not if it doenot contain the field is set to null and if it doen contain then the contents of the section are copied into the kernel data structure. An executable that does not contain an IAT section is executed without any of the checks described in this paper; thus, third party software can be run as-is.

When a system call instruction occurs during the execution of a process, the kernel checks that the address pushed on the stack by the ‘int 0x80’ instruction appears in the IAT information for that process. A system call from an address found in the IAT is allowed to proceed; otherwise, a possible intrusion is signalled.

**using Interrupt Address Tables:**

For finding the system call address we use post-link-time binary rewriting executables. Then e use binary rewriting system for Intel x86 ELF executables. This information is then added to the ELF executable as a new section, the Interrupt Address Table we also use dynamic code or stack to generate system calls.

**Disguising System Call Instructions:**

The problem is that the idefication of the system can is quite easy which is vulnerable for our systems to scan attacks so for that we need to disguise the system call instructions so that it may become harder to identify as for the kernel it is easy to decide which system call is legal which is not because of the existence of all system calls in the instruction address table. By checking the address of each system call in the table so that he find that id it is a false system call or a real system call.

**Insertion of dead and useless code:**

In this method we are inserting randomly choosen instructions sequences in the code so that it only changes the codes contents not it,s semantics. nops and instruction sequences that are functionally equivalent to nops, e.g., ‘add $0, r’, ‘mov r, r’, ‘push r; pop r’, etc., where r is any register; and arithmetic computations into a register r that is not live.in each case we have to make assure that the content is not affected wth our instructions. It is worth noting that some advanced viruses, e.g., encrypted and polymorphic viruses, use a similar mechanism for disguising their decryption engines from detection by virus scanners. The approach can be enhanced using binary obfuscation techniques

**binary obfuscation:**

Binary obfuscation is a technique that aims to shadow the real application code to make it difficult for an external person, who does not have access to your sources, to understand what your program has to do. Obfuscation techniques do not transform your application in an unbreakable one, because with the right effort everyone can gain access to the decrypted data. This is due to the fact that the CPU does not have the ability (yet) to read encrypted data, so you have to deliver unencrypted orders to it. At the end is just an effort to make it hard, for non technical people to follow the execution of your application. It is just a matter of gaining time before your program can be hacked.

+------------------+ File start

| DOS header |

+------------------+

| DOS stub |

+------------------+

| NT header |

+------------------+

| Sections headers | Information over the binary sections

+------------------+

| ... |

+------------------+

| .dummy | Section with code to obfuscate

+------------------+

| |

| |

| |

| ... |

| |

| |

| |

+------------------+ EO

**hindering scanning attacks:**

the acctack code falls when when the system call is hard to identity and then the attacker started to identify the functions which lead to system calls.

We can imagine two classes of such attack we discussed the ways to hinder such attacks. Alternatively, such an attack might scan the program text itself, looking for specific byte sequences. Given a function f in a program P, let If :P be the shortest sequence of instructions (or shortest byte sequence) that uniquely identifies f within P. An attacker might examine his own copy of P, offline, to determine If :P, then craft a scanning attack that searches for this sequence

**CONCLUSION:**

As we all know that in todays modern era security is very important we need to make our system secure from attackers anyone would always prefer a more secure system then the advance one attackers hackers attack it,s their job our job is to protect our system for which wee need to firstly understand the problem then have to solve it in a proper manner. Without understanding the problem it is next to imposible to solve it. We always have to make better plan and design to implement better security on our systems. On operating system level many attacks are been running nowadays but in this paper we are intended to put a light code injection based attacks on systems are commonly performing on system nowadays. And also providing two approaches to escape from this type of attack. First use instruction of addresses tables to allowed only those system calls in kernel which is legitimate to our systems by this we can prevent our system to calling that system call which are trying to make harm for systems. Second is to use different techniques to forbid mimicry attacks that are attempting to execute and identify those system calls which are in program code or in libraries. In this paper the experiment and technique which we are using discussing here is effective about related to such problems.

**Refrences:**

[1] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer,

D. Stefanovic, and D. D. Zovi. Randomized instruction

set emulation to disrupt binary code injection attacks. In

*Proc. 10th ACM Conference on Computer and Communication*

*Security*, pages 281–289, 2003.

[2] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating

system enhancements to prevent the misuse of system

calls. In *Proc. ACM Conference on Computer and Communications*

*Security*, pages 174–183, 2000.

[3] S. Bhatkar, D. C. Du Varney, and R. Sekar. Address obfuscation:

an efficient approach to combat a broad range

[1] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer,

D. Stefanovic, and D. D. Zovi. Randomized instruction

set emulation to disrupt binary code injection attacks. In

*Proc. 10th ACM Conference on Computer and Communication*

*Security*, pages 281–289, 2003.

[2] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating

system enhancements to prevent the misuse of system

calls. In *Proc. ACM Conference on Computer and Communications*

*Security*, pages 174–183, 2000.

[3] S. Bhatkar, D. C. Du Varney, and R. Sekar. Address obfuscation:

an efficient approach to combat a broad range

of memory error exploits. In *Proc. 12th USENIX Security*

*Symposium*, pages 105–120, 2003.

[4] Bulba and Kil3r. Bypassing StackGuard and Stack-

Shield. *Phrack*, 10(56), May 2000.

[5] M. Chew and D. Song. Mitigating buffer overflows

by operating system randomization. Technical Report

CMU-CS-02-197, Electrical and Computer Engineering

Department, Carnegie Mellon University, Pittsburgh, PA

15213, Dec. 2002.

[l] Hoffman, L.J., Modern Methods for Computer Security

and Privacy, Prentice-Hall, Englewood

Cliffs, NJ, 1977.

[2] Martin, J., Security, Accuracy, and Privacy in

Computer Systems, Prentice-Hall, Englewood

Cliffs, NJ, 1973.

[3] Hsiao, D.K., Kerr, D.S. and Madnick, S.E., *s*puter

Security: Its Problems and Solutions, The

ACM Monograph Series, Academic Press, for 1979.

[4] Lampson, B.W., Needham, R.M., Randall, B. and

Schroeder, M.D., "Protection, Security, Reliability",

Operating Systems Review, Vol. 11,

No. 1, Jan. 1977, pp. 12-14.

[5] Linden, T.A., "Operating System Structures to

Support Security and Reliable Software,"

puting Surveys, Vol. 8, No. 4, Dec. 1976.

**Links:**

<https://pdfs.semanticscholar.org/7f5a/934a2048eef2711d9dc48ece5b20ab8046ca.pdf>

<https://www.usenix.org/legacy/event/sec05/tech/full_papers/linn/linn.pdf>