# Launching Return-Oriented Programming Attacks against Randomized Relocatable Executables

Limin Liu* Jin Han† Debin Gao† Jiwu Jing* Daren Zha*

*State Key Laboratory of Information Security
Graduate University of CAS
Beijing, China
{lmliu, jing, zdr}@is.ac.cn

†School of Information Systems
Singapore Management University
Singapore
{jin.han.2007, dbgao}@smu.edu.sg

***Abstract*—Since the day it was proposed, return-oriented programming has shown to be an effective and powerful attack technique against the write or execute only ($W \oplus X$) protection. However, a general belief in the previous research is, systems deployed with address space randomization where the executables are also randomized at run-time are able to defend against return-oriented programming, as the addresses of all instructions are randomized.**

**In this paper, we show that due to the weakness of current address space randomization technique, there are still ways of launching return-oriented programming attacks against those well-protected systems efficiently. We demonstrate and evaluate our attacks with existing typical web server applications and discuss possible methods of mitigating such threats.**

***Keywords*-return-oriented programming; address space randomization; position independent executable;**

## I. Introduction

The write or execute only ($W \oplus X$) policy, which is also called data execution prevention (DEP), is a security feature widely deployed in modern operating systems [1]. It ensures that no memory page is writable and executable at the same time, in order to prevent executing code from a non-executable memory region. It rules out code injection but still allows return-into-libc attacks [2], which tries to reuse the functionality provided by the exploited application. In many cases, such an attack is not effective as the ability of the adversary is restricted by pre-existing functions.

Return-oriented programming (ROP), proposed by Shacham [19], is a general form of traditional return-to-libc attacks. Attacks utilizing ROP chains together existing sequences of instructions that have been previously identified inside the program code or library code. It is shown in previous research [7], [19] that an adversary is able to execute arbitrary functionality and achieve Turing completeness on both x86 and ARM architectures, without injecting any new code inside the application.

A typical defense mechanism – address space layout randomization (ASLR) [4], is believed to be one of the most effective techniques that hamper return-oriented programming [16], [19]. The general ASLR technique randomizes the base addresses of stack, heap, and code segments of a process memory space. As a result, an attacker has to correctly guess the positions of these data structures in order to mount a successful attack. Although prior research [18] has demonstrated that attacks can be launched on address space randomization assuming the base of the shared library is randomized, their conclusion is that ASLR will still be effective when used in combination with position independent executables (PIE) [24], when the address space of both shared objects and executables are randomized.

In this paper, we challenge such strongly protected systems where these three protection mechanisms – $W \oplus X$, ASLR, and PIE – have all been deployed. We find that the current implementations of ASLR (both on Linux [22] and Windows [13]) have a common weakness – all the code instructions are moving together in the program memory space (i.e., only the base address of the code segment is randomized). Thus, if we are able to obtain the address of one instruction correctly (after the code is loaded into memory), we will know the addresses of all the instructions in this executable.

As demonstrated in our paper, there are at least two possible ways of obtaining the randomized address of an instruction under different assumptions. When a format string vulnerability [23] is available in the application, the attack code could be crafted to directly read the address of a specific instruction. When such vulnerability does not exist, one has to randomly guess one instruction address in the code segment. This address can then be used to adjust the addresses of prepared ROP code to efficiently exploit the protected application.

Our paper makes the following contributions:

- We present new ROP attack vectors which are able to successfully exploit systems protected by $W \oplus X$, ASLR, and PIE.
- A detailed analysis is given on utilizing format string vulnerability to obtain information on the stack, and the statistical data on existing format string vulnerabilities are also provided.
- We implement and evaluate our ROP attack with Apache web server and explore the possibility of constructing similar attack code with other applications.

The rest of the paper is organized as follows. Section II

gives the background and the threat model of this paper. Section III introduces the design of our ROP attack utilizing format string vulnerabilities, while Section IV gives the ROP guessing attack. The implementation and evaluation of our attack is presented in Section V. Section VI discusses the limitation of our current attack and also discusses possible mitigation methods. Finally, the concluding remarks appear in Section VII.

## II. Background and Threat Model

### A. Return-Oriented Programming

Return-oriented programming (ROP) was proposed by Shacham in 2007 [19] for the x86 architecture, and then subsequently extended to the SPARC [6], Atmel AVR [12], and other processors [8]. The main idea of ROP is to construct the attack code purely utilizing instructions from existing code (either inside the application or in the linked libraries). These chosen instructions will be chained together in an order designed by the attacker to perform the whole malicious functionality. In the early stage of ROP development, only the instruction sequences which end in a "return" instruction will be chosen to form the ROP code – if the attacker has control of the stack, this will allow control to flow from one sequence to the next.

The way ROP code is constructed makes the instruction stream executed during a ROP attack very different from the instruction stream executed by legitimate programs – it will contain a lot of return instructions, just a few instructions apart and it will pop addresses out of the stack which are not pushed by corresponding "call" instructions. According to these characteristics of ROP attack, various approaches have been proposed by the research community to detect and defeat ROP attacks [9], [11], [15], [16]. However, a recent advancement in ROP [7] has changed the strategy of ROP by adopting different instructions to chain the instruction sequences, thus making most of the existing protection techniques ineffective. In addition, all of those ROP defense mechanisms impose considerable performance overheads and thus have not been widely adopted in modern systems. The most effective and widely deployed technique which is able to mitigate ROP attacks is address space layout randomization [4], as suggested by the literature [18], [16].

### B. Address Space Layout Randomization

Address space layout randomization (ASLR) is a set of techniques [4], [5], [14], [22], [26] which introduce randomness into addresses used by a protected process. This will make a wide class of exploit techniques fail with a quantifiable probability. In addition, the attempts of failed attacks will crash the target task which eases the detection of such attacks. A generic ASLR implementation will randomly relocates the stack, heap, and shared library regions – the base addresses of these memory regions will be randomly changed each time the given program is launched.

The technique that randomly relocates the base of shared libraries in the memory space, has already been wildly deployed on modern operating systems to defend against the return-to-libc attacks. Figure 1 shows the address space layout of an Apache server process under Ubuntu. As shown from line 4 to 9 in Figure 1, the addresses of shared libraries are randomized under Ubuntu. However, the locations of code and data segments (line 1 and line 2 in Figure 1) are not randomized as default.

```
[1]  08048000-080c6000 r-xp   /usr/local/apache2/bin/httpd
[2]  080c6000-080c9000 rw-p   /usr/local/apache2/bin/httpd
[3]  080c9000-08198000 rw-p   [heap]
[4]  b7cd2000-b7cdb000 r-xp   /lib/tls/i686/cmov/libnss_files-2.7.so
[5]  b7cdb000-b7cdd000 rw-p   /lib/tls/i686/cmov/libnss_files-2.7.so
              ⋮                              ⋮
[6]  b7cff000-b7d06000 r-xp   /lib/tls/i686/cmov/libnss_compat-2.7.so
[7]  b7d06000-b7d08000 rw-p   /lib/tls/i686/cmov/libnss_compat-2.7.so
[8]  b7d09000-b7e52000 r-xp   /lib/tls/i686/cmov/libc-2.7.so
[9]  b7e52000-b7e53000 r--p   /lib/tls/i686/cmov/libc-2.7.so
              ⋮                              ⋮
```

Figure 1. Address space illustration of Apache2 under Ubuntu.

Traditionally, the application code itself does not support randomization. The executable files are created by the linker with the assumption that they will be loaded at a fixed address – usually at `0x8048000` as shown in Figure 1. In order to support the randomization of the application code, many operating systems [24] have developed and contributed to the GNU Compiler tool chain, a technique called Position Independent Executable (PIE). Utilizing PIE, binaries are compiled in a way that they could be freely locatable throughout the entire address space of the program. Thus, with the support from both ASLR and PIE, programs are believed to be strongly protected against attacks utilizing ROP [18], since not only addresses of the library code are randomized, but also the instruction addresses of the application itself.

Actually, some implementations of ASLR [5], [14], [22] have also developed their own way of relocating the code segment at run-time. In this paper, we represent PIE as a general type of techniques which enables the system to randomly relocate the program code segment at run-time, instead of differentiating each detailed techniques.

### C. Threat Model

In this work, our attack target is a remote web server which is equipped with PIE technique, and is also protected under both ASLR and $W \oplus X$. In the memory space of the web server process, the base addresses of stack, heap, library code and also the executable code of the server itself have been randomized.

As needed in every attack technique in the literature, we also assume there will be an exploitable vulnerability in the web server program. In Section III, we assume there is an exploitable format string vulnerability, which is not uncommon as we will show with statistical data. In Section IV, we only assume there is a common buffer

overflow vulnerability, which is the same assumption that original ROP attacks have [6], [12], [19].

In this work, we focus on utilizing the binary code of the application itself to construct ROP attacks. Existing ROP attack papers [18], [25] do not consider the strong protection condition where the base address of stack and the executable are randomized (they only consider the base of shared library is randomized), which is different from our work.

## III. ROP Format String attack

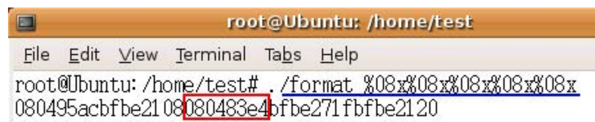
In this section, we will first introduce how to use format string vulnerability to obtain information on the run-time stack. Then we will show how to further utilize such information to launch a ROP attack towards systems protected under ASLR.

### A. *Format String Vulnerability and Information Leakage*

The format string vulnerability (FSV) occurs when unchecked user input is used as the format string parameters in functions that perform the formatting tasks, such as `printf()` in C language. Format string vulnerabilities can be exploited to execute malicious code or print data from the stack or possibly other locations in memory. Although there are various types of format string vulnerabilities and some of them are quite complex, we use a simple illustrative example to demonstrate how such vulnerability can be used to grab the information on the stack, as shown in Figure 2.

```
void foo(char *buf)
{
    printf(buf);
    printf("\n");
}
int main(int argc, char* argv[])
{
    foo(argv[1]);
    return 0;
}
```

(a) An example program (format.c) which contains a format string vulnerability.




(b) An illustrative attack string and the stack information printed.

```
...
0x080483d7 <main+20>:   add    $0x4,%eax
0x080483da <main+23>:   mov    (%eax),%eax
0x080483dc <main+25>:   mov    %eax,(%esp)
0x080483df <main+28>:   call   0x80483a4 <foo>
0x080483e4 <main+33>:   mov    $0x0,%eax
0x080483e9 <main+38>:   add    $0x4,%esp
...
```

(c) The return address of function foo() shown in the debugger.

Figure 2. FSV example code and attack

As illustrated in Figure 2(a), the example function `foo()` directly uses user inputs as the format string. This makes it possible for an adversary to utilize special string like `%08x` to gain the information on the run-time stack (Figure 2(b)). By utilizing GDB[1], we are able to verify that the return address when calling function `foo()` is exactly the same address the attack has obtained, as shown in Figure 2(c).

Format string vulnerability is not uncommon in real-world applications. According to our statistical results analyzed from the CVE database[2], there are considerable amount of format string vulnerabilities which are discovered each year, as shown in Table I. The percentage in the table is the ratio between the number of format string vulnerabilities and the total number of effective[3] vulnerabilities recorded in CVE.

Table I
Statistical data of format string vulnerabilities in CVE

| | 2006 | 2007 | 2008 | 2009 | 2010 |
|---|---|---|---|---|---|
| # of FSV | 66 | 75 | 39 | 25 | 14 |
| Total # of Vul. | 6976 (7039-63) | 6427 (6500-73) | 6969 (7016-47) | 4784 (4814-30) | 4401 (4429-28) |
| Ratio (%) | 0.9461 | 1.1670 | 0.5596 | 0.5226 | 0.3181 |

Further investigation shows that a wide range of programs suffer from format string vulnerabilities, including: web servers such as Apache (CVE-2006-4154, CVE-2006-0150) and Oracle Application Server (CVE-2009-0993); operating systems and virtual machines such as IBM AIX (CVE-2010-1039), Mac OSX (CVE-2010-1376, CVE-2009-2191) and VMware (CVE-2010-1139, CVE-2009-3732); database software such as MySQL (CVE-2009-2446, CVE-2006-3469) and Oracle Database (CVE-2008-5440); and even common libraries such as PHP (CVE-2010-2094, CVE-2007-0909), etc.

All these results indicate that it is reasonable to explore how to effectively attack an application which is assumed to contain a format string vulnerability.

### B. *ROP Attack Utilizing Format String Vulnerability*

As shown from the demonstration in Figure 2, the benefit of exploiting format string vulnerability is the possibility to obtain the instruction address *which has potentially been randomized at run-time*. Once one instruction address has been revealed, the adversary is able to find out where the base address of the code segment is located. Then ROP attack code can be adjusted accordingly in order to successfully exploit this application.

[1]GDB: The GNU Project Debugger, http://www.gnu.org/s/gdb/

[2]The Common Vulnerabilities and Exposures database (http://cve.mitre.org/) is downloadable from National Vulnerability Database (http://nvd.nist.gov/download.cfm).

[3]There are a number of CVE entries rejected each year, although they are still recorded in the CVE database. Thus, as shown in Table I, the total number of effective vulnerabilities equals to the total number of CVE entries minus the number of rejected CVE entries each year.

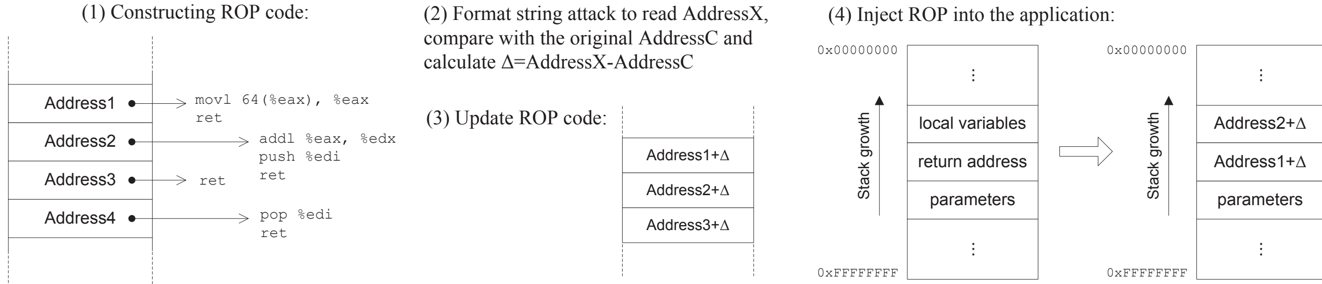



Figure 3. The ROP attack flow for exploiting an application with format string vulnerability

The attack vector is given as follows, which is also illustrated in Figure 3.

1) Constructing ROP attack code (the sequences of ROP addresses) based on the binary code of the target application. The instruction gadgets needed could be found using the Galileo algorithm [19].
2) Exploiting the format string vulnerability to print out one (randomized) return address which has been pushed onto the stack.
3) Using the return address to calculate the difference of the original address and randomized address, and then update the ROP code prepared in the first step accordingly.
4) Launch another format string exploit against the same vulnerability to inject our customized ROP code.

Note that in order for this attack to take effect, the base address of the code segment in Step 4 has to be the same as what has been learned in Step 2. There are two real-world scenarios which could satisfy this condition: a) The attack at Step 2 crash the application's process. However, web servers like Apache may start another child process which has exactly the same address space layout as the previous one. Such a property has also been utilized in existing attacks [20]. b) The format string can be crafted to just exit the vulnerable function without crashing the process. In this case, the same process will still accept the following requests, which contains attack code in Step 4. The second case is confirmed by our attack implementation towards Apache server, which will be further described in Section V.

## IV. ROP Guessing Attack

In many cases, the target web application does not have a format string vulnerability. Since the randomized instruction address cannot be directly obtained, the adversary has to construct ROP code according to a guessed base address. The attack vector is given as follows:

1) Constructing the ROP attack code with instructions in the application binary by guessing a base address of the code segment.
2) Send the attack code to the target web application and wait for the response.
3) If a remote shell is obtained within specified time threshold, then attack succeeds. Otherwise, repeat Step 1 with another base address.

The efficiency of the guessing attack depends on the entropy of the randomization. Due to the design and implementation difference, each ASLR technique may have different randomization entropy. We summarize the entropy of existing ASLR techniques in Table II, based on 32-bit platform setting. Note that although many memory regions are listed in the table, the one that has the impact on our attack is the entropy of the start address of code and data (Bold in Table II). In our current ROP construction, we only utilize the code of the application itself, this means as long as the base address of the code segment is guessed correctly, our ROP attack will succeed. The details of our ROP construction are given in Section V.

According to Shacham et al. [20], PaX only randomize at most 16 bits on a 32-bit Platform. Thus, the ROP guessing attack has a worst success rate of 1/65536, an average success rate of 1/32768. Such a success rate is not secure enough for applications like web servers, which may receive millions of requests each day. Many other attacks which also target on web servers [5] have an even lower success rate compared to ours. Also note that we use ROP code instead of return-to-libc attack to accomplish our exploit tasks, which is different from other proposed guessing attack (such as [20]).

## V. Implementation and Evaluation

In this section, we will first give the ROP construction based on Apache web server. We then verify that we are able to exploit the web application with format string vulnerability when the executable is randomized. Finally, we explore the possibility of constructing such ROP attack on other web applications.

### A. ROP Construction

The goal of our attack is to launch a remote shell which has the same privilege of the exploited web application. Thus, we construct our ROP shellcode in a way that it will

Table II
THE RANDOMIZATION ENTROPY OF EXISTING ASLR IMPLEMENTATIONS

| | Base address of stack | Padding between stack frames | Base address of heap | Padding between malloc()s | **Base address of code & data** | Base address of GOT/PLT |
|---|---|---|---|---|---|---|
| USENIX'03 AO [4] | $2^{26}$ | $2^{8}$ | $2^{8}$ | 25% ($2^{8}$) | $2^{26}$ | |
| SRDS'03 TRR [26] | $2^{29}$ | | $2^{29}$ | | | N/R |
| USENIX'05 [5] | $2^{26}$ | N/R[4] | $2^{26}$ | 30% ($2^{8}$) | N/R | N/R |
| PLDI'06 DieHard [3] | | | | $\leq 2^{3}$ | | |
| PaX ASLR [22] | $2^{16}$ | | $2^{13}$ | | $2^{16}$ | |
| ACSAC'06 ASLP [14] | $2^{28}$ | | $2^{29}$ | | $2^{20}$ | $2^{20}$ |

finally invoke the `execve` system call to launch `\bin\sh`. Different from the ROP shellcode proposed in existing works [6], [19], our shellcode is constructed by completely utilizing the application's own code, instead of utilizing shared library code (such as glibc). It is very difficult to utilize the shared library instructions under our assumption, as our attack targets on systems where the base address of code segment and shared library are both randomized. In order to construct ROP code by using library instructions, besides obtaining the base address of code segment, one has to additionally manage to get the base address of shared library. This would further complicate the attack code and possibly lower down the success rate. Thus, we restrict our ROP code to only include addresses of the instructions within the application executable itself.

Figure 4 gives the details of our ROP code where Apache-2.2.17 is used as the example web application. The base addresses of the stack and code segment have not been randomized in the figure (the code starts at `0x08065668`, and the stack starts at `0xbf9f7f28`). There are 19 words of ROP code in Figure 4: Word 1-5 (from the bottom) set the second argument in `agrv` to be `NULL`; Word 6-8 set `%edx` to the address of `envp` array; Word 9 and 10 set `%ecx` to the address of the `argv` array; Word 11-14 set `%eax` to the system call number `0x0b` (`0xffffffef+0x1c`), and set `%ebx` to the address of "`/bin/sh\0`" and finally, Word 15 calls the kernel interrupt.

Note that some instructions in the shellcode will cause side effects. Such side effects are balanced by additional instructions, which are not shown in Figure 4 (the complete shellcode of Figure 4 is given in Appendix). Especially, we utilize so-called *boring* instructions such as the "leave; ret" instruction sequence in our ROP code. The "leave; ret" sequence is ignored in the Galileo algorithm [19], because such "instruction streams are actually generated by the compiler". However, Shacham [19] also mentions that such "instruction sequences would be useful in crafting exploits". Thus, we modify the Galileo algorithm to also include such instruction sequences when constructing our ROP code.

[4]N/R=Not Reported, i.e., it is mentioned that they carried out the randomization of this region, but the entropy is not given in the paper.

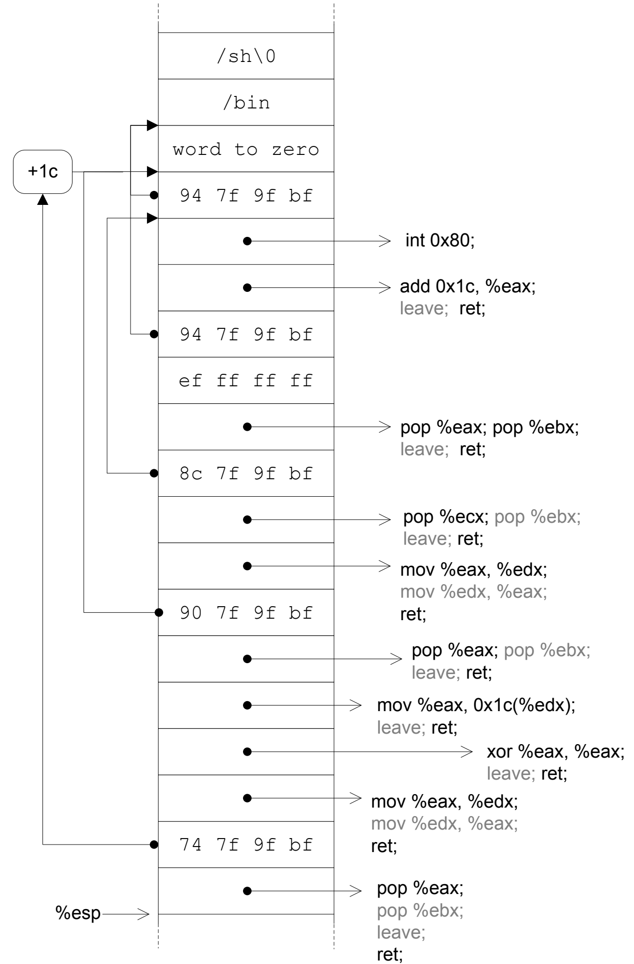



Figure 4. ROP shellcode for Apache-2.2.17

## B. Format String Exploit

In order to verify that our ROP code is able to successfully exploit a web application even when ASLR and PIE are present, we install Apache-2.2.17 and PaX on Ubuntu 10.04 desktop and enable the PaX RANDEXEC[5] feature which is to introduce randomness into the main executable mapping

[5]PaX RANDEXEC, http://pax.grsecurity.net/docs/randexec.txt

addresses. Such settings will make the base addresses of stack, shared library and also the executable code segment change randomly each time when Apache server starts, without the need to recompile Apache.

For the format string vulnerability, we construct a similar one as CVE-2009-4769 in httpdx[6] in Apache-2.2.17. We did not use format string vulnerabilities from the old versions of the Apache server (CVE-2006-4154, CVE-2006-0150) due to insufficient online information.

Our first step attack – the format string exploit embeds `%08x` characters into a `GET` request and sends it to Apache server. Apache server replies with "Invalid URI in request GET ...", and logs the printed stack information into its `error_log`. Our script reads the `error_log`[7] and obtains the randomized return address and save `%eip` address. Based on these two addresses, our ROP code is updated automatically and sent to the Apache server to exploit the same format string vulnerability again. And our result shows that we are able to obtain the command shell.

We rerun our experiments 10 times and draw the conclusion that each time our first step format string exploit will not crash the server process and the stack organization stays exactly the same when the second time our ROP code exploits this format string vulnerability. This shows that our attack is potentially practical – the robustness of the server also makes it less secure in this situation.

### C. *Possibility on Other Applications*

Using the way of constructing ROP code in Section V-A, we investigate other applications to study the possibility of constructing a similar attack. We search the binary code of Apache-2.2.17 and other programs with our modified Galileo algorithm to see if the instruction sequences needed in the attack are available in these programs. The number of instruction sequences found for the five necessary steps in constructing a useful ROP shellcode is given in Table III.

Table III
THE NUMBER OF USEFUL INSTRUCTION SEQUENCES FOUND

| Application | write %eax | write %ebx | write %ecx | write %edx | int 80 / lcall |
|---|---|---|---|---|---|
| apache-2.2.17 (1.3MB) | 142 | 175 | 5 | 45 | 1 |
| doxygen-1.5.8 (4.2MB) | 731 | 897 | 18 | 98 | 2 |
| smbtree-3.3.2 (3.7MB) | 437 | 314 | 11 | 75 | 1 |
| mplayer-1.0.r4 (3.9MB) | 628 | 579 | 13 | 68 | 6 |
| gzip-1.3.12 (56KB) | 8 | 32 | 1 | 3 | 0 |
| iconv-2.13 (50KB) | 15 | 15 | 1 | 4 | 0 |

[6]Format string vulnerabilities in the tolog() function in httpdx 1.4, 1.4.5, 1.4.6, 1.4.6b, and 1.5 allow remote attackers to execute arbitrary code via format string specifiers in a GET request to the HTTP server component when logging is enabled. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-4769.

[7]Here we assume that the adversary has a basic read permission on the server machine so that he is able to open Apache's error log.

Table III shows that it is possible to construct similar ROP attack with programs that are as large as Apache. However, in the executables of small programs, it is very rare to find `int 0x80` or trap-into-kernel instructions (such as `lcall %gs:0x10(,0)` [19]). Thus, it is unlikely to construct similar ROP code for small programs due to the lack of critical instructions.

## VI. DISCUSSIONS

### A. *Limitation of the Current Attack*

Our current attack still has a number of limitations. Firstly, we depends on the format string exploit to give us feedback for not only the randomized base address of code segment, but also the saved `%ebp` value to obtain the randomized stack address, so as to update the corresponding addresses in the ROP code. Shacham [19] suggests that such addresses could be built at runtime by examining `%esp` and operating on it, which would allow the shellcode to be placed at various stack positions without needing retargeting. Such a refinement of our shellcode is left for future work.

Secondly, our current attack requires the adversary to have a basic account on the server machine in order to have the read access to the web server log. This condition makes sure that the adversary is able to get the feedback of the stack information that has been printed out by the format string exploit. However, there could be other ways which the adversary can trick the server to send such information back to client. Study of improving the feedback method of the format string exploit is not within the scope of this paper.

Finally, it is not always able to find format string vulnerability in the target application, especially the vulnerability has to be easily exploitable. Some applications only have such vulnerability in its old versions, which are fixed in the newer version.

Nevertheless, our attack proves that it is still possible to efficiently break through the defense of a system deployed with ASLR where the executable itself is also randomized. However, our result does not mean that ASLR is useless. On the contrary, since our attack has the above limitations, it indicates that ASLR is still an effective defense mechanism against ROP attack.

### B. *Possible Mitigation Techniques*

Some existing techniques proposed can be used to mitigate our attacks. Specifically, approach in [5] permutes all the functions in the code sections in a program at compilation time. If the adversary is not able to access the program executable file on the storage, it will be quite impossible for him to construct ROP code since the positions of required instructions are unknown.

Another defense mechanism towards ROP is G-Free [16]. G-Free is able to eliminate all unaligned free-branch instructions inside a binary executable, and to protect the aligned free-branch instructions to prevent them from being misused

by an attacker. In this way, it is also very difficult to construct ROP attack code as shown in this paper.

There are also a number of research works targeting on automatically discovering format string vulnerabilities [21], [17] or defending against format string exploits [10]. Despite these efforts, it is still a practical problem to eliminate format string vulnerabilities, and new vulnerabilities are continuously being discovered and published every month.

## VII. Conclusion

Address space layout randomization is considered to be a strong defense towards return-oriented programming, especially the executable code segment is also randomized in the program memory space. And indeed it is, as confirmed by our observation. However, in this paper, we have shown that there are still ways of efficiently attack web applications protected under these defense techniques. When a format string vulnerability is available in the program, we can exploit this vulnerability to get the address of the randomized instructions, and then launch a ROP attack against the same vulnerability to obtain the command shell. When only a buffer overflow vulnerability is available, we have to guess the correct base address of the randomized code segment. We demonstrate the first attack with Apache web server and explore the possibility of launching similar attacks on other applications. Although our current attack still have a number of limitations, we believe it can be further improved and be more practical in real-world system scenarios.

## Acknowledgment

This paper was partially supported by National Science Foundation (NSF) China under the agreement 70890084/G021102, Knowledge Innovation Project of Chinese Academy of Sciences under the agreement YYYJ-1013, and National Technology Support Program(NTSP) China under the agreements 2008BAH32B04 and 2009BAH43B03.

## References

[1] A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. Microsoft. 2006-09-26. http://support.microsoft.com/kb/875352/EN-US/.

[2] Solar designer. “return-to-libc” attack. Bugtraq, August, 1997.

[3] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’06, pages 158–168, 2006.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[5] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th conference on USENIX Security Symposium*, 2005.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS ’08, pages 27–38, 2008.

[7] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS ’10, pages 559–572, 2010.

[8] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. In *Proceedings of the 2009 conference on Electronic voting technology/workshop on trustworthy elections*, EVT/WOTE’09, 2009.

[9] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security*, ICISS ’09, pages 163–177, 2009.

[10] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. Formatguard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium*, 2001.

[11] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS ’11, pages 40–51, 2011.

[12] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS ’08, pages 15–26, 2008.

[13] M. Howard. Address Space Layout Randomization in Windows Vista. http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/608315.aspx.

[14] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 339–348, 2006.

[15] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with ”return-less” kernels. In *Proceedings of the 5th European conference on Computer systems*, EuroSys ’10, pages 195–208, 2010.

[16] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC ’10, pages 49–58, 2010.

[17] M. F. Ringenburg and D. Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 354–363, 2005.

[18] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 60–69, 2009.

[19] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, 2007.

[20] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, 2004.

[21] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium*, SSYM'01, 2001.

[22] T. P. Team. address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt.

[23] S. teso. Exploiting format string vulnerabilities. Technical report, Stanford University, September 2001.

[24] A. van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.3, update 3, August 2004. http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf.

[25] Z. Wang, R. Cheng, and D. Gao. Revisiting address space randomization. In *Proceedings of the 13th Annual International Conference on Information Security and Cryptology*, December 2010.

[26] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems*, SRDS '03, pages 260–269, 2003.

## APPENDIX

The complete shellcode of our ROP attack is given as follows:

```
00000000  94 56 06 08 74 7f 9f bf 90 7f 9f bf
0000000C  90 7f 9f bf d7 2c 07 08 b1 dc 06 08
00000018  90 7f 9f bf be 4a 07 08 90 7f 9f bf
00000024  94 56 06 08 90 7f 9f bf 90 7f 9f bf
00000030  90 7f 9f bf d7 2c 07 08 07 76 06 08
0000003C  8c 7f 9f bf 90 7f 9f bf 90 7f 9f bf
00000048  94 56 06 08 ef ff ff ff 94 7f 9f bf
00000054  90 7f 9f bf 3f e4 06 08 90 7f 9f bf
00000060  c0 f9 0a 08 94 7f 9f bf a2 1b f5 bf
0000006C  2f 62 69 6e 2f 73 68 00
```