Revisiting Address Space Randomization

Zhi Wang[1], Renquan Cheng[2], and Debin Gao[2]

[1] College of Information Technology and Science, Nankai University, China
[2] School of Information Systems, Singapore Management University, Singapore

Abstract. Address space randomization is believed to be a strong defense against memory error exploits. Many code and data objects in a potentially vulnerable program and the system could be randomized, including those on the stack and heap, base address of code, order of functions, PLT, GOT, etc. Randomizing these code and data objects is believed to be effective in obfuscating the addresses in memory to obscure locations of code and data objects. However, attacking techniques have advanced since the introduction of address space randomization. In particular, return-oriented programming has made attacks without injected code much more powerful than what they were before. Keeping this new attacking technique in mind, in this paper, we revisit address space randomization and analyze the effectiveness of randomizing various code and data objects.

We show that randomizing certain code and data objects has become much less effective. Typically, randomizing the base and order of functions in shared libraries and randomizing the location and order of entries in PLT and GOT do not introduce significant difficulty to attacks using return-oriented programming. We propose a more general version of such attacks than what was introduced before, and point out weaknesses of a previously proposed fix. We argue that address space randomization was introduced without considering such attacks and a simple fix probably does not exist.

Keywords: Address space randomization, return-oriented programming, software exploit

1 Introduction

Address Space Randomization (ASR) has been proposed as a technique to fight against memory error exploits [2–4]. Most of these techniques obfuscate addresses in memory to obscure the location of code and data objects, including those on the stack and heap, static data, PLT, GOT, and etc. An attacker would then have a hard time finding out the addresses of code and data objects. This in turn makes the result of invalid memory access unpredictable. For example, randomizing the base of the stack and introducing random sized gaps between successive stack frames could make it difficult for an attack to locate or overwrite the return address; randomizing the locations of the PLT and GOT could make it difficult for an attack to access system functions such as `execve()` after subverting the program's control flow and therefore limit what a successful exploit could perform.

However, attacking techniques have advanced a lot since the introduction of address space randomization. In particular, return-oriented programming [14] has made attacks without injected code more powerful, in many cases able to perform arbitrary computation. This raises the question of whether randomizing certain code and data objects is still as effective as what we believed. In this paper, we show that randomizing the base and order of functions in shared libraries and randomizing the location and order of entries in PLT and GOT do not introduce significant difficulty to attacks using return-oriented programming. In particular, we present an attack on a system in which the library base addresses, the order of library functions, and the PLT and GOT are randomized. In the course of presenting the attack, we also detail a few improvements to return-oriented programming to make our attack more effective. We continue to show that a previously proposed fix of encrypting GOT might not work in many cases. We argue that address space randomization was introduced without considering such attacks, and a simple fix probably does not exist.

Note that what we study here is more than returning to randomized lib(c) as shown in a previous work [13]. Besides the attack we propose here being more general, i.e., we consider a system where the order of library functions are also randomized, we strive to study the effectiveness of randomizing various code and data objects rather than proposing a particular attack. We analyze the root cause of attacks using return-oriented programming, point out weaknesses of mitigation techniques in the previous work [13], and argue that randomizing such code and data objects are just ineffective and no simple fix exists. To support our analysis, we evaluate a number of commonly used application programs and show that encrypting GOT is, in fact, not effective in stopping the attack, since there are enough gadgets found in the binary program itself to exercise the attack and returning to libc is not needed.

We caution the readers from drawing from our analysis more than what it deserves. We are not trying to show that address space randomization is not effective in general. On the other hand, since there are many code and data objects that can be randomized, our analysis shows that randomizing some of these does not necessarily improve the system security because of the new attacking technique. Address space randomization is certainly effectively in, e.g., making it difficult for an attack to exploit a vulnerability to subvert the program's control flow. What we show in this paper is that after an attack manages to subvert the program's control flow, the difficulty of causing the program to execute in a manner of his choosing using return-oriented programming is not much affected by randomizing the base and order of functions or location and order of PLT and GOT.

In summary, the paper makes the following contributions.

- Propose and implement a general attack on an address space randomization system where the base and order of library functions and location and order of entries in PLT and GOT are randomized.
- Propose a few improvements to the return-oriented programming to make our attack more effective.

– Analyze limitations of the previously proposed attack mitigation technique of encrypting GOT.
– Discuss on the effectiveness of randomizing the base and order of functions and location and order of entries in PLT and GOT.

The rest of the paper is organized as follows. In Section 2, we outline the background and discuss some related work in this area. Section 3 presents an overview and intuition of our attack. We detail the implementation of our attack in Section 4. Section 5 discusses the limitation of a previously proposed attack mitigation technique and our experimental results on it, and discusses the implications. We conclude in Section 6.

2 Background and Related Work

There are many code and data objects that can be randomized [2–4]. Table 1 presents a summary of the important ones and the specific data to be randomized.

Code and data objects	What to randomize
Stack-resident variables	Base of stack Gaps between stack frames
Heap-resident variables	Base of heap Gaps between heap allocations
Static variables	Order of static variables
Program code	Addresses of function call targets Position independent code
Functions in library	Base of library Order of functions in library Gaps between functions in library
Entries in PLT and GOT	Locations of PLT and GOT Order of entries in PLT and GOT

Table 1. Code and data objects to be randomized

Randomizing these code and data objects is effective in stopping some particular types of attacks or steps in some attacks. In this paper, we try to analyze the effectiveness of randomizing some of these data in making attacks difficult. In particular, our analysis shows that randomizing functions in library and entries in PLT and GOT is ineffective. We support this by presenting our general attack on an address space randomization system and analyzing an attack mitigation technique previously proposed.

To understand how these randomization helps in making attacks difficult, we briefly describe the two steps an attack usually needs to perform. First, it needs to find a way to exploit the vulnerability to subvert the program's control flow. Second, it needs to cause the program to execute in a manner of his choosing. Traditionally, the first step could be done by overflowing a buffer on the stack and overwriting a return address, although many other techniques, e.g., heap [8] and integer overflows [18] and format string vulnerabilities [16], could be used.

The second step can be done by executing injected code [12] or performing a return-to-libc attack.

Address space randomization [2–4] and a variant of it [17] are proposed to make both steps discussed above difficult. For example, in order to overwrite a return address on the stack to subvert the program's control flow, an attack needs first to locate the return address. If the base of the stack is randomized, the location of the return address is no longer the same on different executions of the same program and therefore the attack will be difficult. A brief summary of the randomizing techniques to make it difficult to subvert the program's control flow follows.

- Introducing shadow stack for buffer-type variables;
- Randomizing the base of the stack and heap;
- Introducing random sized gaps between successive stack frames and heap allocations;
- Avoiding calls using absolute addresses by transforming them into function pointers.

Address space randomization can also make it difficult for an attack to perform arbitrary computation after the attack subverts the program's control flow. For example, making memory spaces non-writable or non-executable could stop injected code execution. Randomizing functions in the binary and shared library could make return-to-libc attacks difficult. Here is a summary of randomizing techniques to make this step difficult.

- Making certain memory spaces non-writable or non-executable;
- Randomizing the order of functions in the binary and shared libraries;
- Introducing random sized gaps and inaccessible pages between functions in the binary and shared libraries;
- Randomizing the order of static variables;
- Randomizing the location of PLT and GOT;
- Randomizing the order of entries in PLT and GOT;
- Uses position independent code in the program.

In this paper, we assume that an attack has successfully subverted the vulnerable program's control flow (first step), and try to evaluate how effective address space randomization is in making the second step difficult, i.e., in making it difficult for the attack to perform arbitrary computation.

Our attack uses the idea of return-oriented programming [14, 6]. Return-oriented programming fits the requirement of the attack well because it does not need to execute any injected code. Only a large number of short instruction sequences from either the original program or libc is to be executed in order for the attack to perform arbitrary computation. However, our attack is more challenging than return-oriented programming on a normal (non-randomized) machine in that the addresses of the short instruction sequences are randomized and unknown to the attacker. Although return-oriented programming has been extended to a number of different environments [5, 9–11, 7], it is non-trivial how it can be applied on address space randomization systems.

Perhaps the work to surgically return to randomized libc [13] is the closest to our work in this paper. In this work, Roglia et al. introduced an attack on address space randomization assuming the base of the libc library is randomized. The attack surgically finds the address of a libc function by reading entries in PLT and GOT using return-oriented programming. The attack we present in this paper uses the same strategy, but differs in that it also assumes that the order of library functions are randomized. Roglia et al. also proposed an attack mitigation technique of encrypting the GOT. In this paper, we argue that such a technique might not work on programs where enough gadgets are found in the program binary itself and libc is not needed for the attack. We demonstrate this by analyzing a few commonly used application programs and show that an attack on them indeed does not require the use of libc. In general, this paper is not just about introducing an attack on address space randomization, but to study the effectiveness of randomizing certain code and data objects, and to argue that randomizing them is ineffective to defend against attacks using return-oriented programming, and a simple fix does not exist.

The effectiveness of address space randomization on 32-bit architectures has been analyzed previously [15]. In this work, a brute force attack is proposed to guess the libc text segment offset in order to perform a return-to-libc attack. Experiments show that such an attack is effective on a 32-bit system where the vulnerable service automatically restarts after crashing. Our attack is different from this attack in that we derandomize the addresses in an efficient way without brute forcing. Therefore, our attack has a wider application on systems where counter-measures are in place to fight against brute force attacks.

3 Attack on Address Space Randomization

As shown in Section 2, there are many code and data objects that can be randomized to make different attacks or attack steps difficult. Although return-oriented programming [14] has made attacks without injected code more powerful, in many cases able to perform arbitrary computation, intuitively it does not work well on address space randomization systems because the locations of gadgets are randomized and hard to be found.

In this section, however, we show that randomizing the base of the library, order of library functions, entries in PLT and GOT is ineffective in defending against attacks using return-oriented programming. We show this by presenting an attack on an address space randomization system where we assume that position independent code is not in use in the binary program. This assumption is valid in most existing computing systems because recompilation is needed to generate position independent code. We show that our attack is able to execute arbitrary computation after subverting the control flow of the program. This attack uses the same strategy of the one presented by Roglia et al. [13]. However, here we assume that the order of library functions is randomized whereas Roglia et al. only considers the randomized base address.

In the rest of this section, we first give an intuition of the attack we propose and an overview of the steps involved. In Section 4, we detail the implementation of the attack and a few improvements we introduce to make return-oriented programming more effective in our attack.

3.1 Attack intuition

As many memory pages are made non-writable or non-executable in an address space randomization system, our attack tries to use existing code in the system to perform arbitrary computation. A typical way of performing such an attack is to use return-to-libc attacks to transfer control to system function `execve()`. Recall that we assume that the first step of the attack to subvert the control flow of the program, see Section 2, has been done. Therefore, the most important next step is to locate the address of a system call in existing code (e.g., in libc) and then transfer control over there.

Randomizing base address of the library and order of library functions Randomizing the base address of libc and the order of libc functions are definitely effective in making our attack more difficult, since the address of these function has been randomized and cannot be pre-computed in our attack.

Randomizing entries in PLT and GOT PLT (procedure linkage table) and GOT (global offset table) play crucial roles in resolution of library functions, and therefore is a potential target of our attack. As shown in Figure 1, GOT stores the address of libc functions, while PLT contains entries that jump to the addresses stored in GOT.

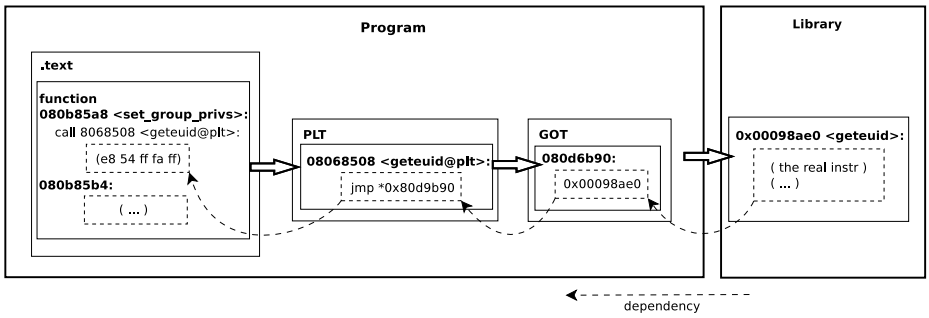

Fig. 1. PLT and GOT in a dynamically linked ELF executable

The dependency between randomizing PLT/GOT and randomizing library base address and functions was well documented — if an attacker knows the location and offsets of PLT, then the address of libc functions can be found even if the base address of libc and order of libc functions are randomized [4].

We have seen the dependency between randomizing libc and randomizing PLT/GOT because addresses of libc functions are used in PLT/GOT. By the same token, entries of PLT/GOT are used by other parts of the program, in particular, by `call` instructions in the code segment. If an attack can locate such `call` instructions in the program, theoretically the target of the `call` would reveal the location and offset in PLT/GOT, too. This analogy can also be seen from Figure 1.

Another way to look at such an attack is that no matter how well code and data objects are randomized, the randomized object would need to be accessible by the original program anyway to enable execution of the program. Addresses of libc functions are randomized, but the randomized addresses are used in PLT/GOT to allow libc functions to be called; by the same token, PLT and GOT can be randomized, but the randomized addresses are used in `call` instructions to allow functions to be called, too. If our attack is able to locate the `call` instructions and find out the target of the call, we can find the address of libc functions indirectly.

3.2 Attack overview

To demonstrate the chain of dependencies, we propose our attack to perform arbitrary computation when the binary program does not make use of position independent code, i.e., when the attacker has access to the vulnerable program for static analysis. In such a scenario, the attacker can easily locate the `call` instructions by disassembling the code segment. However, finding out the (randomized) target of the call still remains nontrivial since it requires a memory read operation to be executed. Recall that 1) we assume that memory pages are non-writable or non-executable, and therefore executing injected code is not an option; 2) libc function addresses have not been found, and therefore return-to-libc is not an option either.

However, with the advances of return-oriented programming [14], such an attack becomes possible. Return-oriented programming fits the requirement of the attack well because it does not need to execute any injected code. Instead, it can make use of short instruction sequences from the original program (not the libc since the randomized libc addresses have not been found yet) to perform the read operation (and some others; see Section 4). Figure 2 shows the steps involved in our attack.

After the control flow of the program is subverted (our assumption), our return-oriented programming code will first read the target of a `call` instruction whose address is known by static analysis of the vulnerable program. After that, we locate the address and offset through PLT and GOT. Once the entry in PLT and GOT is located, we read the entry to find out the corresponding libc function, and eventually we can use the short code sequences inside libc. In the end, the address of the libc function can be used to obtain a shell for arbitrary computation by making a system call. Note that our attack works well when the order of library functions is randomized, which a previously proposed attack does not consider [13].

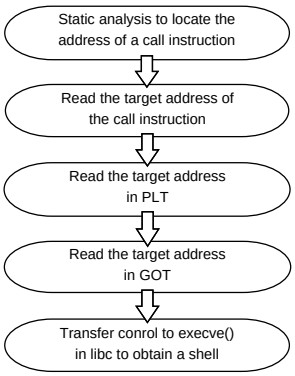

Fig. 2. Overview of our attack

4 Attack Implementation

As discussed in Section 3.2, there are a few steps involved in a successful attack, and each step requires some instructions to be executed. In this section, we first explain in more details what instructions are needed in each step, and then present a realization of executing these instructions using a few improvements to the return-oriented programming. We demonstrate our attack with an example on `apache-2.2.15`.

4.1 Instructions needed to be executed in our attack

The first step in our attack is to find the static address of a useful `call` instruction in the code segment of the vulnerable program. There are typically many `call` instructions in the code segment, and what we need is 1) one that calls a libc function; and 2) the corresponding libc function makes a system call. We need the second requirement in order to make sure that we can later make use of the system call to execute `execve()` for arbitrary computation. The one that we choose is `call geteuid` at `0x80b85af` in apache-2.2.15 (see Figure 3). Note that many other `call` instructions could be used.

```
080b85a8 <set_group_privs>:
 80b85a8:     55                push   %ebp
 80b85a9:     89 e5             mov    %esp,%ebp
 80b85ab:     53                push   %ebx
 80b85ac:     83 ec 34          sub    $0x34,%esp
 80b85af:     e8 54 ff fa ff    call   8068508 <geteuid@plt>
 80b85b4:     ...
```

Fig. 3. call instruction in the code segment

Finding target address of the call instruction As shown in Figure 3, the target address of the `call` instruction is represented as an offset (`0xfffaff54`) of the address of the next instruction (`0x080b85b4`). Therefore, in order to obtain the target address of the `call` instruction (`0x08068508`), our attack needs two instructions, i.e., a memory read instruction (at an address of our choosing) to read the offset, and an add instruction to add the offset to the address of the next instruction (static).

Finding jump target address in PLT Every entry in PLT has 3 instructions that correspond to 16 bytes; see Figure 4. What we are interested in the jump target in is the first instruction, assuming that the program has been executing for a while and lazy linking has already initialized the address of the GOT entry in the first instruction. To find the jump target (`0x08d06b90`), we need another add instruction to find the address of the jump target (offset of 2 bytes at `0806850a`) and another memory read instruction to read the jump target address.

```
08068508 <geteuid@plt>:
 8068508:     ff 25 90 6b 0d 08     jmp    *0x80d6b90
 806850e:     68 20 17 00 00        push   $0x1720
 8068513:     e9 a0 d1 ff ff        jmp    80656e0 <_init+0x30>
```

Fig. 4. Entry in PLT

Finding the address of the libc function in GOT This step is simple, as the jump target found in PLT contains exactly the address of the libc function; see Figure 5. Therefore, we need only a memory read instruction here.

```
080d6b90 <_GLOBAL_OFFSET_TABLE_+2972>: e0 8a 09 00
```

Fig. 5. Entry in GOT

Making a system call Once the address of the libc function (`geteuid`) is found, we can make a system call by transferring control to an instruction inside the libc function. Figure 6 shows the instructions inside `geteuid`, in which the fourth instruction `call %gs:0x10` is the new system call instruction in Linux. We first initialize four register values (`eax, ebx, ecx, edx`) and then transfer control to this instruction. So our attack in this step simply needs register initiation instructions.

```
00098ae0 <geteuid>:
 98ae0:     55                          push   %ebp
 98ae1:     89 e5                       mov    %esp,%ebp
 98ae3:     b8 c9 00 00 00              mov    $0xc9,%eax
 98ae8:     65 ff 15 10 00 00 00        call   *%gs:0x10
 98aef:     5d                          pop    %ebp
 98af0:     c3                          ret
```

Fig. 6. System call in libc

4.2 Finding gadgets to realize the instructions needed

In this subsection, we outline how the instructions needed in our attack are realized by return-oriented programming [14]. The idea of return-oriented programming is to use gadgets (short code sequences ended by `ret`, or by `jmp <reg>` [6]). Note that in our attack, these gadgets have to be found in the vulnerable program except in the last step after the libc function address has been found. This makes our attack more challenging than return-oriented programming in general where useful gadgets can be easily found in the large libc library.

Since the vulnerable program is usually relatively small when compared to the libc library, we might not be able to locate the gadgets we want. We propose and use a few techniques to expand the set of useful candidate gadgets. We do not further discuss how the last step of our attack can be implemented by finding useful gadgets in libc since it has been well discussed in the return-oriented programming paper [14].

Alternative instructions There could be multiple different instructions that serve what we need in the operations. Table 2 shows some candidate gadgets of different instructions for the same purpose needed in our attack. Note that they are just some examples, and each of them could have different variations, e.g., by using different registers.

Operations	Useful gadgets
Memory reading	`<mov (%eax), %eax; ret;>`
Addition	`<add %ebp, %ebx; ret;>`
	`<lea (%eax, %ecx, 1), %eax; ret;>`
Register writing	`<pop %eax; ret;>`
	`<xchg %eax, %edx; ret;>`

Table 2. Useful gadgets with alternative instructions

Combination of instructions Besides using gadgets of different instructions, we can also combine different instructions (their corresponding gadgets) together to realize the intended operation. For example, `<or (%eax), %ebx; ret;>` `or`'s the value at a memory address (specified by `eax`) with another register (`ebx`). It serves the purpose of memory reading if `ebx` happens to be zero. Even if `ebx` is

not zero, this gadget can be combined with a register writing to set `ebx` to be zero first. Table 3 gives some examples of such combinations.

Operations	Useful gadgets
Memory reading	`<register writing>` `<or (%eax), %ebx; ret;>`
Addition	`loop: <inc %eax; ret;>`
Register writing	`<mov $const, %eax; ret;>` `<lea ($const), %eax; ret;>` `<addition>`

Table 3. Useful gadgets by combining instructions

Instructions with side-effects Some instructions in a gadget might have no effect in the execution context or might have side effects that can be reversed by other gadgets. Although these instructions (and the corresponding gadgets) make our analysis more complicated, taking them into consideration helps us find more useful gadgets. For example, in searching for gadgets to pop data from the stack to a register, we only managed to find `<pop eax; ret;>` and `<pop ecx; ret;>` directly from `apache-2.2.15`. After analyzing instructions with some side-effects, we managed to find `<pop ebx; pop ebp; ret;>` and `<pop edx; push eax; std; dec ecx; ret;>` with one and three instructions with side-effects in the middle, respectively.

4.3 Attacks on apache and other programs

With the techniques discussed in Section 4.2, we search the binary code of `apache-2.2.15` and other programs to see if gadgets needed could be found using the Galileo algorithm [14]. The number of gadgets found for different operations are presented in Table 4.

Programs	Memory reading	Addition	Register writing
`apache-2.2.15` (695 KB)	2	7	34
`vsftpd-2.2.2` (116 KB)	1	3	47
`bind-9.7.0` (486 KB)	3	1	17
`sendmail-8.14.3` (806 KB)	1	4	14
`mplayer-1.0~rc3` (4 MB)	5	19	117
`firefox-3.6.3` (50 KB)	0	1	13

Table 4. Number of gadgets found

Table 4 shows that we manage find the needed gadgets from `apache`, `vsftpd`, `bind`, `sendmail`, and `mplayer`, while relatively small programs, e.g., `firefox`[3], may not provide enough useful gadgets.

[3] Firefox is a large program, but its binary file, /usr/lib/firefox-3.6.3/firefox-bin (under Ubuntu-10.04), is only of 50 KB as most functionality is provided in libraries.

To try out our attack on `apache-2.2.15` on a real system, we downloaded the address space randomization proposed by Bhatkar et al. and migrated the code to a PAX-enabled Ubuntu 10.04 desktop computer. We configure the system such that base address of the library, order of library functions, PLT and GOT are randomized. We then use `gdb` to overflow a buffer of `apache-2.2.15` on the stack with our attack code. The attack successfully creates a shell for arbitrary computation. Appendix A shows the shell code that we use in this attack. Since it is possible to find the needed gadgets from various programs as shown in Table 4, we believe that our attack can be generalized to be applied on other vulnerable programs. We leave this as our future work.

4.4 Discussions of our attack

What we propose is a more general attack which works even when the order of library functions is randomized, which is different from a previously proposed attack [13].

Other considerations of our attack In the discussions above, we have not considered a level of indirection address space randomization might have introduced, namely converting direct function calls to indirect ones with function pointers. Our attack works in the same way when function pointers are used; in fact, the attack could even be simplified in some cases because offsets might not be used in indirect calls.

Limitations of our attack There are a few limitations of our attack. First, we assume that the control flow of the vulnerable program can be subverted. This might not be true as address space randomization could make such subverting very difficult. However, this assumption does not hinder our analysis less important because a security system should not rely on the single point of protection and should try to make attacks difficult even when the first line of defense fails. Second, we assume that the attacker has access to the vulnerable program to do static analysis and position independent code is not in use. Our attack relies on this assumption because we wouldn't be able to locate the `call` instruction should this assumption be invalid. Third, we might not be able to find enough useful gadgets from the vulnerable program. Although we have shown programs meeting our attack requirement, it remains future work to study other ways of finding useful gadgets to generalize our attack.

Extension of our attack The idea of our attack could be extended to make stack randomization ineffective, if instructions like `mov eax, esp` could be found by using return-oriented programming. We tried using the Galileo algorithm [14] to search for it, but could not find one in our experiments. Theoretically, this is possible especially when searching on various sections that are marked executable, e.g., `.plt`, `.text`, `.fini`, `.rodata`, `.eh_frame_hdr`, and `.eh_frame`. We leave this as future work.

5 Possible Mitigation Techniques and Discussions

Roglia et al. proposed a few mitigation techniques to defend against attacks that dereference and overwrite GOT [13], which include using position independent code, self-randomization of the program, and encrypting GOT. Although such techniques could defend against our attack presented as well, we try to ask a deeper question: is address space randomization weak in randomizing GOT only and therefore becomes effective once the mitigation techniques are in place, or is it true that randomizing some of the code and data objects (e.g., base and order of library functions) is simply ineffective when return-oriented programming is used in an attack?

Before we try to answer this question, we first revisit our attack presented in Section 3 and Section 4 and see if exploiting GOT is the only way for the attack to succeed. The answer is definitely not. We try to derandomize the address of libc functions simply because the library has a larger code base which could be analyzed offline and usually contains more useful gadgets for return-oriented programming. However, in many cases, all an attack wants is simply to be able to make a system call (with values of the attacker's choice on a few registers), which might be possible with only gadgets from the vulnerable program itself without making use of the library. We perform an analysis on some commonly used application programs by using the Galileo algorithm [14] and our improvements on it (see Section 4.2) to search for gadgets that allow an attack to make a system call. Results (see Table 5) show that some programs, such as the vulnerable version of Ghostscript [1], could be attacked by only gadgets from the program.

In an attack using return-oriented programming with gadgets in the program binary only, even fewer gadgets could be required. For example, to execute the `execve()` system call, we only need to write four registers (`eax`, `ebx`, `ecx`, `edx`) and then execute the system call instruction. Only these two categories of instructions (and the corresponding gadgets) are needed. Table 5 shows the number of gadgets found for a few application programs.

Programs	Register writing	syscall (int80 or call *%gfs:0x10)
`gs-8.61` (11 MB)	34	130
`mencoder-4.3.2` (8.7 MB)	47	5
`emacs-23` (11 MB)	143	15
`qemu-0.11.1` (2.1 MB)	23	10
`qmake 2.01a` (3.8 MB)	27	4

Table 5. Number of gadgets found in some large programs

This shows that GOT is actually not the most important weaknesses in address space randomization in view of attacks using return-oriented programming. Rather, because address space randomization was proposed well before return-oriented programming was introduced, it was not designed to defend against return-oriented programming and therefore it is not surprising that the random-

ization of some of the code and data objects is simply not effective to defend against return-oriented programming. We argue that the randomization of base and order of library functions and the location and order of entries in PLT and GOT are typical examples. Mitigation techniques like encrypting GOT does not actually make address space randomization secure against return-oriented programming.

6 Conclusion

In this paper, we demonstrate our attack on randomizing the base address of library, order of library functions, and entries in PLT and GOT with return-oriented programming under the assumption that the attacker has a copy of the vulnerable program for static analysis. Besides introducing this more general attack and proposing improvements to return-oriented programming to make the attack more effective, we also evaluate an attack mitigation technique previously proposed. Results show that dereferencing GOT is actually not a necessary step in the attack, and therefore encrypting GOT does not make address space randomization secure against return-oriented programming.

References

1. CVE-2008-0411, "Ghostscript (8.61 and earlier) zseticcspace() Stack-based Buffer Overflow Vulnerability".
2. PaX. `http://pax.grsecurity.net`, 2001.
3. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security 2003)*, 2003.
4. S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security 2005)*, 2005.
5. 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 2008)*, 2008.
6. 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)*, 2010.
7. 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 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE09)*, 2009.
8. Solar Designer. JPEG COM marker processing vulnerability. 2000. `http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability`.
9. 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 2008)*, 2008.

10. R. Hund, T. Holz, and F. C. Freiling. Returnoriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security 2009)*, 2009.
11. T. Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-University Bochum, Germany, 2009.
12. Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 1996. `http://www.phrack.com/issues.html?issue=49&id=14`.
13. G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009)*, 2009.
14. 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 2007)*, 2007.
15. H. Shacham, M. Page, B. Pfaff, E. 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 2004)*, 2004.
16. Scut/team teso. Exploiting format string vulnerabilities. 2001. `http://team-teso.net`.
17. J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Symposium on Reliable and Distributed Systems (SRDS)*, 2003.
18. M. Zalewski. Remote vulnerability in ssh daemon crc32 compensation attack detector. 2001. Bindview.

A Shell Code of our Attack

```
00000000  e8 01 05 08 d0 85 0b 08  3d fe 06 08 bf bf bf bf
00000010  3d fe 06 08 bf bf bf bf  ac c6 0c 08 ac c6 0c 08
00000020  3d fe 06 08 bf bf bf bf  3d fe 06 08 bf bf bf bf
00000030  ac c6 0c 08 ac c6 0c 08  ac c6 0c 08 ac c6 0c 08
00000040  ac c6 0c 08 ac c6 0c 08  ac c6 0c 08 ac c6 0c 08
00000050  51 63 0a 08 e8 01 05 08  58 f4 ff bf 51 63 0a 08
00000060  66 b4 08 08 bf bf bf bf  e8 01 05 08 08 80 04 08
00000070  3d fe 06 08 bf bf bf bf  51 63 0a 08 e8 01 05 08
00000080  60 f4 ff bf 51 63 0a 08  66 b4 08 08 bf bf bf bf
00000090  e8 01 05 08 60 f4 ff bf  51 63 0a 08 a8 02 05 08
000000a0  5c f4 ff bf c2 85 06 08  64 f4 ff bf bf bf bf bf
000000b0  e8 01 05 08 5c 82 04 08  3d fe 06 08 bf bf bf bf
000000c0  bf bf bf bf 64 f4 ff bf  bf bf bf bf 2f 62 69 6e
000000d0  2f 73 68 00
```