A Sensitivity Analysis of Common Operating Systems to ROP Attacks

Marco Prandini and Marco Ramilli

DEIS - Università di Bologna, Viale del Risorgimento 2, 40136 Bologna, Italy

Abstract. Return Oriented Programming (ROP) is a well know technique used by attackers to build the last generation of stack-based attacks. ROP uses small code sequences ("gadgets") to invoke code from the stack, but bypassing the NX bit security protection, allowing attackers to control the execution flow. This paper analyzes some widespread operating systems, profiling the gadgets that can readily be used, and deducing what kind of payloads they allow to build. Understanding which gadgets are usable from the attacker's perspective is of great practical importance to devise countermeasures to the possible attacks.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 Return Oriented Programming

Return-oriented programming is an attack technique that recently attracted significant attention from the scientific and professional communities for its effectiveness against most up-to-date systems. Buchanan et Al. in [2] describe how to turn good code into bad code using an alternative way of parsing it; this technique was introduced in 2007 by Shacham and later named: Return Oriented Programming (ROP). The ROP technique uses code misalignment to forge new instructions from data already loaded into the memory. This technique is made possible by the so called IA32 "density". The IA32 architecture has so many opcodes that almost every sequence of bytes could be interpreted as a valid IA32 instruction. Consequently, by parsing a binary code not from the "natural" entry point but from an address misaligned by one or more bytes with respect to it, it is highly likely to find usable sequences of instructions. Shacham on his paper on ROP proved that such a misalignment produces a Turing complete language, which gives to the programmer the ability to write any possible code. Each instruction sequence ending in the IA32 instruction `RETN` is called a *gadget*. Gadgets are remarkable because they allow the attacker to regain the control of the execution flow after performing some computation. If the attacker knows where to find gadgets, he can start a jump/return chain to execute arbitrary code. The first gadget is invoked by placing its address on the stack (%ebp + 4), then the attacker gets the control flow back since every gadget ends with a `RETN`. By modifying the next return address on the stack the attacker could redirect the control flow to another gadget and so on and so forth until he wants to end the code execution.

Fig. 1 represents a possible stack layout without address space layout randomization. The first jumping address (`0xbadbeef`) is placed in `%eip` (`%ebp` + 4) giving the control to the first gadget which, in this specific scenario, is a simple `RETN` instruction that transfers the control flow to the address found in next 4 bytes: a pointer

Prandini M. and Ramilli M..
A Sensitivity Analysis of Common Operating Systems to ROP Attacks.
DOI: 10.5220/0004099300850092
In *Proceedings of the 9th International Workshop on Security in Information Systems* (WOSIS-2012), pages 85-92
ISBN: 978-989-8565-15-0
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

to RegCreateKeyEx. RegCreateKeyEx keeps some inputs parameters through POP instructions from the stack. Each needed parameter is given to the running function through the stack. Eventually a 4 bytes stack padding is added to compensate each further POP/PUSH instructions that might misalign the %SP register. Once a function has finished a RETN is invoked giving back the control to next address in the stack, which in Fig. 1 specific case points to the WriteFile function.

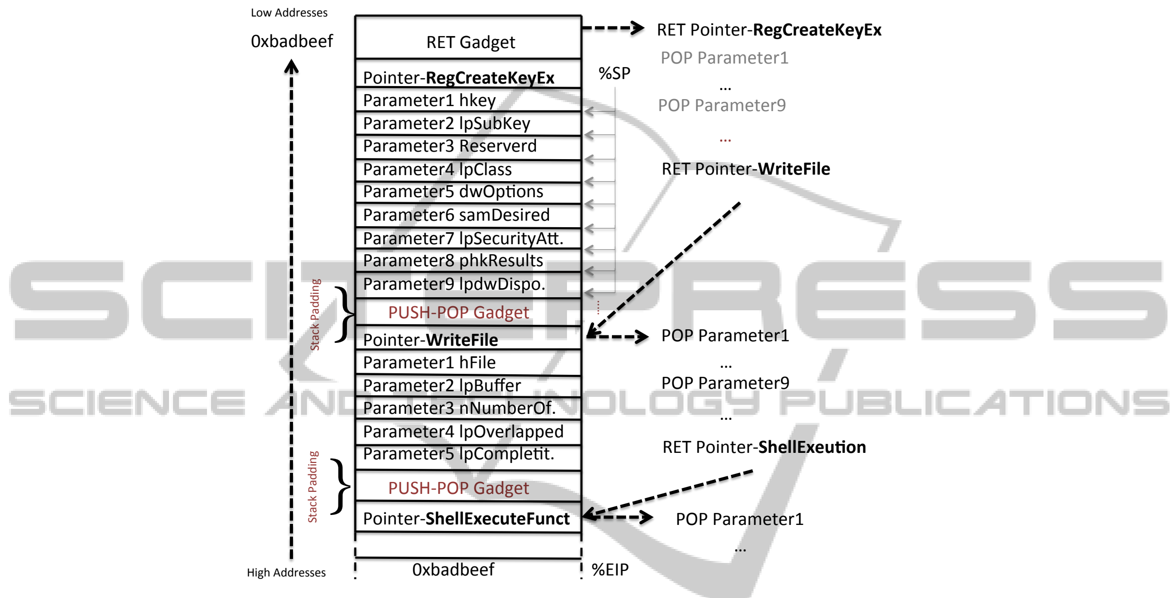

Fig. 1. A possible stack layout of a gadget chain.

Ralf Hund et Al. in [7] present the design and the implementation of the first rootkit which, by using the ROP technique, fully automates the process of constructing malicious instruction sequences reusing memory data. Hovav Shacham in [3] shows a new concept of attack which combining a large number of short instructions is able to build ROP gadgets allowing arbitrary computation on the target machine. Thomas Dullien et Al. in [5] described the current algorithms used for finding gadget into the memory and an automated implementation of a framework to find out gadgets into every possible file. Checkoway et Al. in [3] went even beyond the previous research by studying a ROP technique without any return statement. Their attack makes use of certain instruction sequences that behave like a return, which occur with sufficient frequency in large libraries on (x86) Linux and (ARM) Android to allow creation of Turing-complete gadget sets. Because they do not make use of return instructions, they can circumvent several recently proposed classes of defense against ROP.

Even though much research has been conducted on the topic, no comprehensive and implemented defense mechanism has been proposed so far. Kaan Onarlioglu et Al. in [9] proposed G-Free, a compiler-based approach that represents a first practical solution. G-free is able to eliminate all unaligned free-branch instructions inside a binary executable, and it is able to protect the aligned free-branch instructions to prevent them

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

from being misused by an attacker. Even if the project is implemented and working, very few developer communities are using it, and in any case it is a compiler-based solution which assumes that every developer uses a ”patched” compiler to build their applications. This is a far-fetched assumption, and for sure cannot be readily applied to the installed base of vulnerable software. Michalis Polychronakis et Al. described a ROP payload detection technique that makes use of speculative code execution, where they explore the space of admissible code flows for binaries that already exists in the address space of a targeted process according to the scanned input data, and identify the execution of valid ROP code at runtime. This approach is pretty expensive in term of performance, and a tool implementing it, if not continuously kept well up-to-date, could cause a sensitive amount of false positives.

This paper studies how the common attacking technique called Return Oriented Programming could be adopted to compromise brand-new operative systems. In section 2 we analyze some of the most countermeasures to fight ROP, in 3 we study the native gadgets in common operative systems. By adopting a misalignment of 15 bytes we extract all the available gadgets from four selected operative systems and we study their frequencies. In section 4 we shows some of the real payloads freely available in exploit-db [1]. Section 5 and 6 concludes the whole study by comparing the found gadgets to the analyzed payload.

SCIENCE AND TECHNOLOGY PUBLICATIONS

2 ROP Countermeasures

Researches put their efforts in finding good ways to fight ROP malware, for example Davi et Al. [8] And Chen et Al [4] [10] implemented a threshold system able to count how many sequences of instruction followed by RETN are present in a given executable, once the threshold is reached the security mechanism alerts the user about that. Another direction has been to look for violations of last-in, first-out invariants of the stack data structure that call and return instructions usually maintain in normal benign programs. Francillon Et Al. [6] implemented shadow return address in hardware stack such that only call and return instructions can modify the return address stack. Davi et al. claim that it is possible to extend their ROP defender with a frequency measurement unit to detect attacks with return-less ROP. The idea is that pop-jump sequences are uncommon in ordinary programs, while “returnless” ROP invokes such a sequence after each instruction sequence. Kanjie Lu [8] propose a ”conversion tool” able to transform the most advanced ROP-Based payloads into equivalent non-ROP payloads, which can subsequently be analyzed by standard malware analysis tools such as sand boxes and decompilers, but is still not able to prevent ROP attacks in real time.

All these countermeasures could be classified as detection mechanisms since the goal of these techniques is to detect possible Return Oriented Programming attacks; none of these mechanisms would prevent ROP attacks. Currently, the two most widely deployed OS-built defense mechanisms are not effective against ROP attacks, since ROP was devised exactly for circumventing the NX bit/DEP protection and the address space layout randomization (ASLR) [1], has already been bypassed in more than a way [12],[11],[13].

[1] http://www.exploit-db.com/

3 Gadgets Available in Common OSes

We consider four of the most used Linux based operative systems: BackTrack 5 which is very different from normal OS because it offers an unusually large number of executables and hacking/security tools; Debian 6 which is one of the most widely deployed distributions, and is conversely very minimal; Fedora Core 15 which is another standard distribution for desktop purposes; Ubuntu Server 11.10 which is one of the most used operative systems in server environments.

Longld at vnsecurity.net implemented an open source tool able to seek gadgets inside a given executable, called ROPEME [2]. We used a modified version of ROPEME in order to scan all the available executables and to extract all the available gadgets from them.

There are significant quantitative differences between the analyzed distributions. BackTrack 5 is the one with more ready-to-be-used gadgets followed by Fedora Core 15, Ubuntu Server 11 and Debian 6. The number of gadgets depends on several factors such as: dimension of the executable, number of libraries embedded into the executable, number of executables and executable operations. BackTrack is the one with the highest number of executables and the one with the highest number of embedded libraries. This explains why it contains almost 160,000 gadgets while desktop/server distributions fare between 40,000 and 120,000.

SCIENCE AND TECHNOLOGY PUBLICATIONS

An interesting parameter to be analyzed is the frequency of predetermined and well known operations inside the gadgets. It gives us a measure to compare what the gadgets do and if they can be used to compose common payloads. In other words this parameter let us understand how good the chances are of rebuilding known malware, which was probably neutralized by the appearance of NX and ASLR, as a ROP program that uses the available gadgets. The left side of Fig.2 shows the total count of `PUSH`, `POP`, `ADD`, `MOV`, `INC`, `XOR`, `MUL`, `DIV`, `XCHG`, `SUB`, `LEA`, `CALL` and `JMP` instructions over the found gadgets. BackTrack 5 classifies first, meaning that, in addition to having the highest number of gadgets, it also has the highest number of useful and easy-to-use ones. It is followed by Fedora Core 15, Ubuntu Server 11 and Debian 6.

The presence of `PUSH` and `POP` operations within the gadgets increase the payload complexity, because each operation like these that changes the stack pointer calls for the introduction of a 4-bytes padding into the injected payload. For this specific reason the attackers prefer to use gadgets with as few of these instructions as possible. The second parameter we measured is the frequency with which the different registers are used. Gadgets that do not change registers (besides the ones needed for the payload-specific computation) allow the construction of simpler chains; otherwise, save/restore operations are to be added. The right side of Fig 2 shows the register usage by gadgets per operating system. Obviously the register selection is not on the developer's hands, but in the compiler hands, this might explain why different operative systems behave in a very similar way. `%eax` is the most used register for almost all the analyzed operating systems. The frequency of the second and third most used registers differs from distribution to distribution. The graph in Fig. 2 (right) shows a commonplace asymmetry in general-purpose registers usage, with `%eax` and `%ebx` prevailing over `%ecx`

[2] http://www.vnsecurity.net/2010/08/ropeme-rop-exploit-made-easy/

and %edx. Another common characteristic is the frequency of usage of %ebp, which is higher than the frequency of usage of any other registry (excluding %eax and %ebx). Here Fedora Core makes an exception, showing instead a more marked usage of %esi and %edi.

From an attacker's perspective the number of used registers is a very interesting index since it represents the kind of building block available to code the payload's algorithm. Different register usages lead to different attacker choices, for example a high number of gadgets operating on many different registers could allow a more complex and compact coding of the same algorithm than what could be done in an environments were it is likely to find only gadgets that operate on few registers.

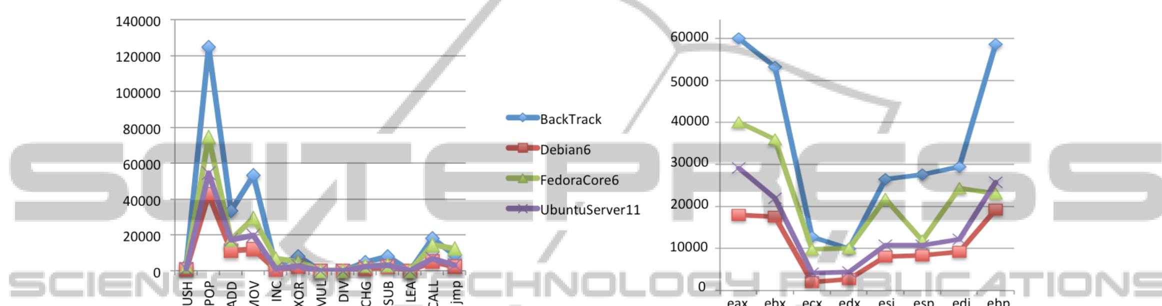

Fig. 2. Assembly operations (left) and CPU registers (right) usage frequency in gadgets.

4 Analysis of Real Payloads

In order to apply the previous results to real-life applications, we need to study the same analyzed parameters from real payloads. Once we classify the basic operations payloads carry out, and the most frequently used registers, we can compare these findings with the gadgets characteristics of the analyzed operating systems, to infer whether they are likely vulnerable to ROP-based rewriting of the analyzed payloads. Through automatic scripts we gathered 312 payloads from exploit-db.com and Google, by looking for predetermined names and fixed strings that commonly appear in gadget-based malware. Once we collected the desired number of payloads we computed the same kind of stats already gathered for gadgets.

Listing in fig. 4 shows an example of a gadget based exploit taken from exploit-db. The python code is made for Aviosoft Digital TV Player Professional 1.x and exploits a stack based buffer overflow through a .plf file. For each grabbed exploit we analyzed the core structure (in the example is the content of the "rop" variable) by counting keywords in the comments next to the gadget addresses. Fig 4 shows the results computed over the full set of found malwares. On the left hand side we show the frequency of the common assembly instructions found in the analyzed payloads. POP is the most common instruction. It is followed by ADD, MOV, XCHG and PUSH. No MUL, DIV and JMP instructions were found into the analyzed payloads. On the right hand side we show the CPU register usage frequency. %eax, %ecx and %ebx are the most used registers in real payloads, while %esi, %esp and %edi are less used registers.

```
import struct
file = ’adtv_bof.plf’
totalsize = 5000
junk = ’A’ * 872
align = ’B’ * 136
# aslr, dep bypass using pushad technique
seh = struct.pack(’<L’, 0x6130534a) # ADD ESP,800 # RETN
rop = struct.pack(’<L’, 0x61326003) * 10 # RETN (ROP NOP)
rop+= struct.pack(’<L’, 0x6405347a) # POP EDX # RETN
rop+= struct.pack(’<L’, 0x10011108) # ptr to &VirtualProtect()
rop+= struct.pack(’<L’, 0x64010503) # PUSH EDX # POP EAX # POP ESI # RETN
rop+= struct.pack(’<L’, 0x41414141) # Filler (compensate)
rop+= struct.pack(’<L’, 0x6160949f) # MOV ECX,DWORD PTR DS:[EDX] # POP ESI
rop+= struct.pack(’<L’, 0x41414141) * 3 # Filler (compensate)
rop+= struct.pack(’<L’, 0x61604218) # PUSH ECX # ADD AL,5F # XOR EAX,EAX # POP ESI # RETN 0C
rop+= struct.pack(’<L’, 0x41414141) * 3 # Filler (RETN offset compensation)
rop+= struct.pack(’<L’, 0x6403d1a6) # POP EBP # RETN
rop+= struct.pack(’<L’, 0x41414141) * 3 # Filler (RETN offset compensation)
rop+= struct.pack(’<L’, 0x60333560) # & push esp #  ret 0c
rop+= struct.pack(’<L’, 0x61323EA8) # POP EAX # RETN
rop+= struct.pack(’<L’, 0xA13977DF) # 0x00000343-> ebx
rop+= struct.pack(’<L’, 0x640203fc) # ADD EAX,5EC68B64 # RETN
rop+= struct.pack(’<L’, 0x6163d37b) # PUSH EAX # ADD AL,5E # POP EBX # RETN
rop+= struct.pack(’<L’, 0x61626807) # XOR EAX,EAX # RETN
rop+= struct.pack(’<L’, 0x640203fc) # ADD EAX,5EC68B64 # RETN
rop+= struct.pack(’<L’, 0x6405347a) # POP EDX # RETN
rop+= struct.pack(’<L’, 0xA13974DC) # 0x00000040-> edx
rop+= struct.pack(’<L’, 0x613107fb) # ADD EDX,EAX # MOV EAX,EDX # RETN
rop+= struct.pack(’<L’, 0x60326803) # POP ECX # RETN
rop+= struct.pack(’<L’, 0x60350340) # &Writable location
rop+= struct.pack(’<L’, 0x61329e07) # POP EDI # RETN
rop+= struct.pack(’<L’, 0x61326003) # RETN (ROP NOP)
rop+= struct.pack(’<L’, 0x60340178) # POP EAX # RETN
rop+= struct.pack(’<L’, 0x90909090) # nop
rop+= struct.pack(’<L’, 0x60322e02) # PUSHAD # RETN
nop = ’\x90’ * 32
```

Fig. 3. An example of gadget-based exploit.

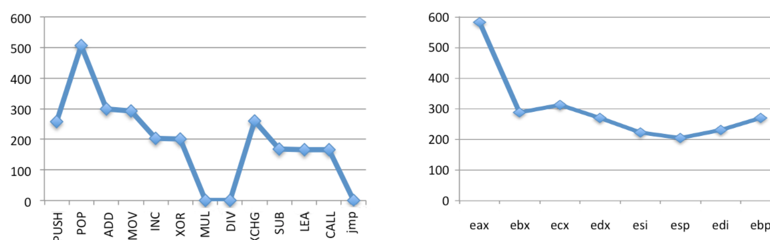

Fig. 4. Assembly operations (left) and CPU registers (right) usage frequency in payloads.

5 Discussion

Real world payloads work by using `POP`, `ADD`, `MOV`, `XCHG` and `PUSH` IA32 assembly instructions. Operating systems offer gadgets that mostly make available `POP`, `ADD` and `MOV` instructions.

Real world payloads work by using `%eax`, `%ecx` and `%ebx` registers. Operating

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

systems offer gadgets acting on `%eax`, `%ebx`, `%edi`, and `%ebp`.

Comparing those results we observe that many payloads using the `XCHG`, `LEA`, `SUB` and `CALL`, which are difficult to find in unmodified operating systems, could be ported onto them with some difficulty. Conversely, payloads which mainly use instructions such as `PUSH`, `POP`, `MOV`, `ADD`, `INC` could be used with high success in unmodified operating systems, but this is a pattern seldom found in typical payloads.

Regarding registers, we notice that the usage distribution of `%eax` and `%ebx` in typical payloads is suitable to match the characteristics of all operating systems, whereas an attack to Fedora Core could be slightly more difficult to mount since this distribution shows few gadgets operating on the `%esp` and `%ebp` registers which are required by typical payloads.

6 Conclusions

In this paper we analyzed how real-world payloads could successfully attack stock operating systems. The research follows a simple statistic analysis comparing the IA32 assembly instructions set and registers set found in 312 real-world payloads to the IA32 assembly instructions set and registers set found into the gadgets offered by four common operating systems in the GNU/Linux family. Comparing these parameters we observed that there might be two different kinds of payloads: the ones making intensive use of `XCHG`, `LEA`, `SUB` and `CALL`, and the ones making intensive use of texttPUSH, `POP`, `MOV`, `ADD`, and S`INC`. Our research shows that the analyzed operating systems could be more easily attacked by the second kind of payload. In other words if a user runs an unmodified operating system without any additional software on it, he might adopt countermeasures able to detect/prevent only the smallest set of payloads: the one using PUSH, POP, MOV, ADD and INC. On the other side we observed that both of the payload sets make intensive use of a registry pattern followed by all but one operative systems. Only Fedora Core 15 differs, exhibiting very few gadgets that manipulate two registers that are commonly used by payloads. This might significantly increase the security of this kind of systems since very few payloads can work without using the CPU registers that Fedora Core 15 gadgets does not allow to manipulate.

References

1. 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 - Volume 12, SSYM'03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.
2. 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, New York, NY, USA, 2008. ACM.
3. 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, New York, NY, USA, 2010. ACM.

SCIENCE AND TECHNOLOGY PUBLICATIONS

4. P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In Prakash and Gupta [10], pages 163–177.
5. T. Dullien, T. Kornau, and R.-P. Weinmann. A framework for automated architecture-independent gadget search. In Proceedings of the 4th USENIX conference on Offensive technologies, WOOT'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.
6. A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In Proceedings of the first ACM workshop on Secure execution of untrusted code, SecuCode '09, pages 19–26, New York, NY, USA, 2009. ACM.
7. R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In Proceedings of the 18th conference on USENIX security symposium, SSYM'09, pages 383–398, Berkeley, CA, USA, 2009. USENIX Association.
8. K. Lu, D. Zou, W. Wen, and D. Gao. derop: removing return-oriented programming from malware. In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11, pages 363–372, New York, NY, USA, 2011. ACM.
9. 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, New York, NY, USA, 2010. ACM.
10. A. Prakash and I. Gupta, editors. Information Systems Security, 5th International Conference, ICISS 2009, Kolkata, India, December 14-18, 2009, Proceedings, volume 5905 of Lecture Notes in Computer Science. Springer, 2009.
11. 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, New York, NY, USA, 2004. ACM.
12. R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In Proceedings of the Second European Workshop on System Security, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.
13. H. Xu and S. J. Chapin. Address-space layout randomization using code islands. J. Comput. Secur., 17(3):331–362, Aug. 2009.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS