DROP: Detecting Return-Oriented Programming Malicious Code

Ping Chen[1], Hai Xiao[1], Xiaobin Shen[2], Xinchun Yin[2], Bing Mao[1], and Li Xie[1]

[1] State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University, Nanjing 210093
{chenping,xiaohai}@sns.nju.edu.cn, {maobing,xieli}@nju.edu.cn
[2] College of Information Engineering, Yangzhou University, Yangzhou Jiangsu 225009, China
xcyin@yzu.edu.cn

Abstract. Return-Oriented Programming (ROP) is a new technique that helps the attacker construct malicious code mounted on x86/SPARC executables without any function call at all. Such technique makes the ROP malicious code contain no instruction, which is different from existing attacks. Moreover, it hides the malicious code in benign code. Thus, it circumvents the approaches that prevent control flow diversion outside legitimate regions (such as $W \oplus X$) and most malicious code scanning techniques (such as anti-virus scanners). However, ROP has its own intrinsic feature which is different from normal program design: (1) uses short instruction sequence ending in "ret", which is called gadget, and (2) executes the gadgets contiguously in specific memory space, such as standard GNU libc. Based on the features of the ROP malicious code, in this paper, we present a tool DROP, which is focused on dynamically detecting ROP malicious code. Preliminary experimental results show that DROP can efficiently detect ROP malicious code, and have no false positives and negatives.

1 Introduction

Return-Oriented Programming (ROP) is a technique which chains together existing instruction streams ending in a "ret" instruction, and then it perform arbitrary, Turing-complete computation without code injection. The instruction streams can be extracted from existing library/binary (e.g., standard GNU libc). Now it is not only available on the x86 platform [40], but also can be mounted on SPARC architecture [14].

ROP technique can be used to rewrite existing malicious code, and eventually become serious threats when used to compromise the computer, which we called as ROP attack. Similar to traditional attacks, such as remote code-injection attack, ROP attack leverages the software vulnerability to launch an attack. However, there are significant differences between ROP attack and traditional attack. ROP attack uses existing library/binary, and ROP malicious code contains only immediate data words and addresses which are pointed to the short instruction sequences in libc, rather than the instructions on which traditional attack relies. ROP malicious code breaks the assumption that preventing the attacker from executing injected code is sufficient to protect a computer, which is at the core of anti-virus software, and other defenses like Intel and AMD's $W \oplus X$ protections. In addition, although ROP attacks have common feature with traditional return-into-libc attacks [23, 27, 29], it is more difficult to defend ROP

A. Prakash and I. Sen Gupta (Eds.): ICISS 2009, LNCS 5905, pp. 163–177, 2009.

attacks for the following reasons: traditional return-into-libc attack uses libc functions, which can be wrapped or removed by the maintainers of libc. By contrast, ROP attack uses short instruction sequences, and each sequence is typically just two to five instructions length. It is non-trivial to remove the short instruction sequences, which exist in the libc or other library/binary widely.

Security tools have arm race with attack techniques. Attacks often use the software vulnerabilities to achieve their goals. Based on this observation, vulnerability detecting tools are leveraged to protect the vulnerability, such as buffer overflow and format string. Although several tools [13, 17, 18, 19, 39] can effectively defend a lot of existing vulnerabilities, none of them can assure that they have prevented all the software bugs. Besides, zero-day attacks become more serious than before, and they can compromise thousands of hosts in a few minutes. It is not sufficient that we only focus on detecting vulnerabilities, because it is too late to defend zero-day attacks. And we need to dynamically monitor the execution behavior of zero-day attacks. Based on the reasons mentioned above, there are a lot of defense tools which aim at detecting the malicious code according to its characteristics. Take remote code-injection attack for example, early works [22, 24, 34, 38, 42] aim at extracting the signature of the shellcode by pattern-based analysis, and the signature is the single, contiguous code sequence. As attackers are employing advanced evasion techniques such as polymorphism to circumvent the defense tools. Some works [15, 25, 32, 43] are further specific to these polymorphic attacks. For example, SigFree [43] is an attack blocker which audits whether the network packet contains instruction sequence and can be leveraged to detect some polymorphic shellcode. However, ROP attack will be resilient to all these defense tools mentioned above, as it has significant difference from traditional attacks, which assumes that malicious code contains instructions to achieve malicious purpose. Thus, these works will be blind to ROP payloads. Other tools [35, 36, 41, 45] use network emulation to detect the remote-injection code, and identify the execution behavior of polymorphic shellcode. Currently, network emulation solutions depend on discovery of instruction sequence, which does not exist in ROP malicious code, since ROP malicious code is totally combined by constant value and instruction address in libc or other existing library/binary. Thus it will be ineffective for ROP attacks. Moreover, based on the observation that most remote-injection attack will execute the code which is injected into the memory by the attacker, another detecting method $W \oplus X$ is used to detect shellcode. These techniques are deployed by the PaX [1] project as a patch for Linux. However, ROP attacks execute existing binary code in program, so it will not be detected by $W \oplus X$.

Although ROP attacks may circumvent many existing defense tools, and hide their malicious behavior in benign code. We find that ROP attack has its intrinsic feature. (1) ROP attack uses gadget ending in "ret" instruction which is used to jump to the next gadget, and the number of the instructions in the gadget is often less than five. By contrast, in benign program design, the pairs of "ret" instruction and "call" instruction represent the prologue and epilogue of the function. (2) "ret" instructions contiguously executed in ROP attacks and they pop up the addresses of the gadgets in existing library/binary. Whereas in normal program, "ret" instructions pop up the addresses which are not contiguously located in the same existing library/binary, that is to say, the distribution of addresses are dispersed. Based on the two differences between ROP attack and normal

programs, we develop a tool named DROP, which dynamically detects ROP attack by checking whether the execution trace deviates from the normal execution route.

Our paper makes three major contributions:

- We select several gadgets from glibc-2.3.5 and leverage these gadgets to rewrite 130 x86 shellcode on milw0rm [28] by ROP technique.
- We statistically analyze a number of normal applications and ROP malicious code, and we point out the factors which represent the feature of ROP malicious code.
- We develop an effective tool to detect the ROP attack, with the best of our knowledge, our tool is the first one on detecting ROP attacks.

The rest of this paper is organized as follows: ROP attacks are described in section 2. In section 3, we present an overview of DROP. The design and implementation of DROP is illustrated at section 4. Section 5 provides the evaluation results of our tool. Section 6 examines its limitations, followed by a discussion of related work in section 7. Finally, section 8 concludes our work.

2 ROP Attack

In this section, we first describe the design of ROP malicious code. In practice, we extract several gadgets from glibc-2.3.5, and rewrite 130 x86 shellcode from milw0rm [28] by using these gadgets. Based on the experience of writing ROP malicious code, we point out the feature of ROP attacks.

2.1 Design of ROP Malicious Code

We extract 30 gadgets from glibc-2.3.5 based on the algorithm of finding useful instruction sequence [40]. All these gadgets contain no more than five instructions. We ignore the following "boring instructions" [40]. (1)*"pop ebp"* and *"leave"*, these two instructions ending in "ret" cannot be used in ROP shellcode. (2)*"unconditional jump"*, we ignore the code sequence *"jmp XXX; ret;"*, instead, we use the gadget *"pop %esp; ret;"* to perform the unconditional jump by changing the value of %esp.

Based on the gadgets we find in glibc-2.3.5, we rewrite 130 Linux x86 shellcode from milw0rm [28]. Adopting the ROP techniques proposed by Hovav Shacham [40], we also develop additional techniques to rewrite the ROP malicious code, and these techniques can improve the design of ROP malicious code.

- *Data Segment*: We put the "unconditional jump" gadget after the padding bytes('0x41') in the shellcode, and define the data segment next to the "unconditional jump". Then unconditionally jump to the next gadget which is close to the data segment. Just like in C Programming, we declare variables and constants at the beginning of the function, the data segment is used to store the temporary values or constant arguments of the system call. This technique avoids complicated calculation of the memory address used in ROP shellcode, especially when there are a lot of temporary values and constant arguments used by original shellcode.

– *Constant Value*: There are often some immediate values in shellcode, such as the system call number. In ROP shellcode, we cannot store them in memory directly, because it will bring NULL bytes to the shellcode. Alternatively, we store its negative values in the memory. Take "11(0xb)" for example, we store its negative value "-11(0xfffffff5)" in the memory, and use the gadget *"pop %edx; ret;"* to load -11 to %edx , then leverage the other two gadgets *"xor %eax, %eax; ret;"* and *"sub %edx, %eax; ret;"* to get the original immediate value 11.
– *Shortest Gadget Sequence*: We try to use the shortest gadget sequence to rewrite the original code. For example, if we want to load a value to memory, the gadget sequence mentioned by Hovav Shacham is *"pop %eax; ret; mov %eax, 0x18(%edx); ret"* [40], it contains two gadgets. By contrast, we use following gadget instead to achieve the same functionality. The difference is that we need to store the value subtracted by 10 in shellcode, and then pop it to *%ecx*.

```
pop     %ecx
add     $0xa,%ecx
mov     %ecx,(%edx)
ret
```

Figure 1 shows one example of the 130 ROP shellcode we rewrite. Figure 1 (a) shows the original shellcode, and Figure 1 (b) shows the ROP shellcode. These two kinds of shellcode have the same function: obtaining a command shell from which the attacker can control the compromised machine. In this example, glibc-2.3.5 is mapped at address 0x03000000, program stack is mapped at address 0x4fffff00, and in practice, we assume these addresses have already been obtained by the attacker.

```
0x31 , 0xc0,  0x50 ,                    /*  xor %eax, %eax;
                                          push %eax
                                        */
0x68,  0x2f,  0x2f,   0x73, 0x68,  /* push$0x68732f2f;*/
0x68,  0x2f,  0x62,  0x69, 0x6e,  /* push$0x6e69622f;*/
0x89,  0xe3,                        /* mov %esp, %ebx;*/
0x50,                               /* push %eax; */
0x53,                               /* push %ebx;*/
0x89,  0xe1,                        /* mov %esp ,%ecx;*/
0x31,  0xd2,                        /* xor %edx, %edx;*/
0xb0,  0x0b,                        /* mov $0xb, %al;*/
0xcd,  0x80;                        /* int 0x80;*/
0x00
```

(a)

```
0x9e,  0x7a,  0x03,  0x03 , /* xor %eax, %eax; ret; */
0xe8,  0x7f,  0x02,  0x03,  /* pop %edx; ret; */
0x0c,  0xff,  0xff,   0x4f,
0x10,  0x80,  0x02,  0x03, /*mov %eax, 0x18(%edx);ret;*/
0xe8,  0x7f,  0x02,  0x03, /* pop %edx; ret; */
0xf5,   0xff,  0xff,   0xff , /* -11 */
0x9b,  0xa0,  0x06,  0x03, /* sub %edx, %eax; ret; */
0x0d,  0xb1,  0x06,  0x03, /* pop %ebx; ret; */
0x38,  0xff,   0xff,  0x4f,  /* address of "/bin//sh" */
0xe7,  0x7f,  0x02,  0x03,  /* pop %ecx; pop %edx; ret;*/
0x20,  0xff,  0xff,  0x4f,
0x24,  0xff,  0xff,  0x4f,
0xf5,  0xda, 0x08,  0x03,  /* int 0x80; ret;*/
0x38,  0xff,  0xff,  0x4f,
0x12,  0x34,  0x56,  0x78,
0x2f,  0x62,  0x69,  0x6e,
0x2f,  0x2f,   0x73,  0x68, 0x00
```

(b)

Fig. 1. Example code: (a) original shellcode, and (b) the ROP shellcode

In Figure 1, we can see that except constant data *"/bin//sh"*, ROP malicious code is in the different shape from original code. The original code contains instructions, whereas ROP malicious code is consisted of the address of gadget and immediate data

within libc. In addition, ROP malicious code leverages the gadgets ending in a "ret" instruction which pops up the address of the next gadget.

2.2 Features of ROP Malicious Code

Return-oriented Programming malicious code relies on existing code (e.g., libc) and contains no instructions. The organizational unit of a return-oriented attack is the gadget. Each gadget is an arrangement of words on the stack, and these words point to instruction sequences and immediate data words. When the gadget is invoked, it accomplishes several well-defined task, such as a load or an arithmetic operation [37].

Based on the practical experience of writing ROP malicious code, we find the feature of ROP malicious code as follows:

- ROP chains together gadgets (often contain no more than 5 instructions) which are already existing in the memory space, and each of these gadgets ending in "ret".
- ROP malicious code utilizes the contiguous gadget sequence.
- ROP technique hides the malicious code in benign code, as it only contains the immediate data or address value.

Formula in Table 1 is used to represent ROP malicious code. In this Formula, we define:

Definition 1 (Candidate Gadget). *Candidate Gadget refers to the instruction sequence ending in "ret". We defined the number of instructions in a Candidate Gadget as G_size. Candidate Gadget Set is briefly represented as G[1...n], G[i] represents the i_{th} Candidate Gadget in G set.*

Definition 2 (Contiguous Candidate Gadget Sequence). *Contiguous candidate gadgets are defined as the gadgets occur one after the other and they pop up the address within the same library/binary memory space. The Contiguous Candidate Gadget Sequence contains the contiguous candidate gadgets, and it is represented as S[1...k], S[i] represents the i_{th} Candidate Gadget in S set. The length of contiguous candidate gadget sequence is defined as S_length, and Max(S_length) represents the maximum value of S_length.*

Table 1. Expressions represent the ROP malicious code

G_size	$= sizeof(G[i])$ *If* $Min\_Addr <= G[i].Addr <= Max\_Addr \&\& G[i] \in G;$
S	$= \{S[i] \mid S[i].G\_size, S[i+1].G\_size <= T0 \&\& S[i]^1, S[i+1]^1 \in G\};$
S_length	$= \{length \mid length = sizeof(S)\};$
ROP	$= Assert(Max(S\_length) >= T1);$

[1] S[i] and S[i+1] are contiguous gadgets.

In Table 1, *Min_Addr* / *Max_Addr* is the start/end address of existing library/binary, where the gadgets are extracted from. *G_size* and *Max(S_length)* are the two factors which represent the feature of ROP malicious code. *T0* and *T1* are the thresholds of the *G_size* and *Max(S_length)*, respectively. To detect ROP malicious code, we need correctly choose the value of *T0* and *T1*.

3 Overview

Based on the differences between ROP malicious code and normal program, we implement a defense tool "DROP" to detect ROP malicious code. Based on the thresholds of *G_size* and *Max(S_length)*, DROP monitors the program dynamically, intercepts the "ret" instruction, chooses the "ret" instruction which pops up the address in libc, and then checks whether the maximum length of contiguous candidate gadget sequence is more than *T1* and each gadget has no more than *T0* length. If so, DROP raises an alarm that the process executed contains ROP malicious code.

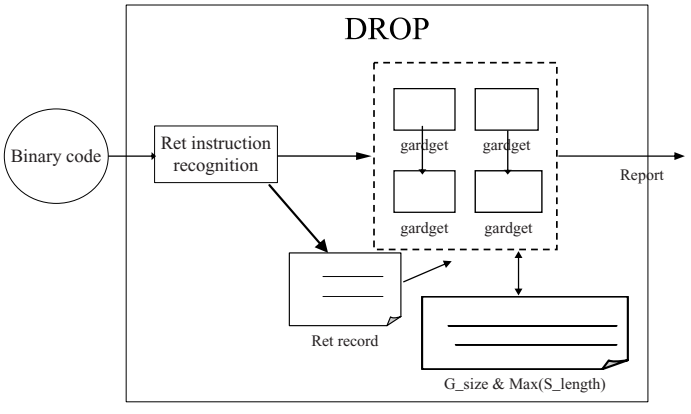

Fig. 2. Architecture of DROP

Figure 2 shows the architecture of our system. First, we recognize the "ret" instruction and determine whether it pops up the address within libc. If so, we record the address poped up by "ret" instruction. Second, we record the size of each *Candidate Gadget* (*G_size*). And we also record the length of contiguous gadget sequences (*S_length*). By referencing the thresholds of *G_size* and *Max(S_length)*, we check whether ROP malicious code exists. Note that our system currently inspects the gadgets in the libc, it can be extended to other existing library/binary such as Linux Kernel.

4 Implementation Details

Our system is implemented on dynamical binary instrumentation tool Valgrind-3.4.0 [30]. DROP dynamically instruments the binary code and does statistical analysis to determine whether the execution route breaks the thresholds of *G_size* and *Max(S_length)*, which are two main factors to separate the ROP malicious code from normal program.

Figure 3 shows the flow chart of DROP. First, DROP leverages Valgrind Core to translate the binary code into intermediate language VEX. Second, DROP recognizes the "ret" instruction represented by VEX. Third, DROP records the address poped up by "ret" instruction and checks whether the address is in libc. Then DROP counts the length of candidate gadget, which equals to the number of instructions between two

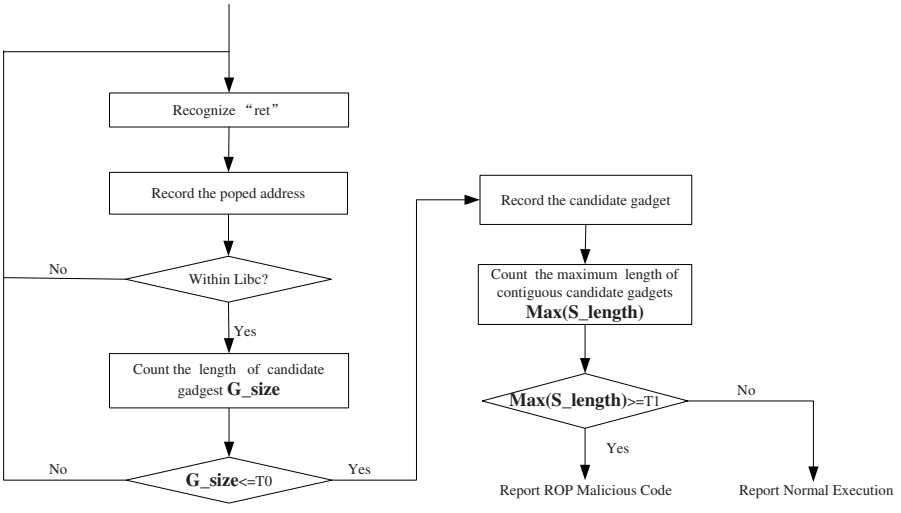

Fig. 3. Flow chart of DROP

"ret" instructions, and selects the candidate gadget whose length is no more than *T0*, that is to say, *G_size<=T0*. Finally DROP checks whether there are more than *T1* contiguous gadgets, in other words, *Max(S_length) >=T1*. If the binary execution meets the feature of ROP malicious code, DROP will report that there exists ROP malicious code.

There are two challenges to detect the ROP malicious code.

– *Ret Instruction Recognition.*
First we need to recognize the "ret" instruction and then record the address poped up by it. Valgrind translates binary code into intermediate language VEX, and Table 2 shows "ret" instruction represented by VEX in 32 bits architecture. There are four

Table 2. Ret Instruction Represented in VEX

```
[1]PUT(60) =0x804838A:I32;
[2]t3 =LDle:I32(t4);
[3]t5=Add32(t4,0x4:I32);
[4]PUT(16) =t5;
```

VEX statements which represent the "ret" instruction. The first statement is used to store the address of "ret" instruction in %eip(60), the second statement is used to pop up the address from the top of stack, then the third and fourth statements are used to regulate the value of %esp by adding 4 to it. In intermediate language VEX, *PUT* is used to write the value to the register, and *LDle* is used to load the value to the memory. We found the representation of "ret" instruction in VEX has following feature: (1) Using *LDle* to read the value from the top of stack; (2) Using the *Add32* expression , whose first operand is the same as the operand of *LDle* and second operand is 0x04; (3) The result of addition statement is the same as the

right operand of *PUT* statement, and the register of *PUT* statement is %esp(16). We instrument the second VEX statement, which pops up the address from the top of stack, to record the address. When we identify the "ret" instruction, we check whether the address popped up by the "ret" instruction is in libc.

– *Contiguous candidate gadgets Recognition* . DROP recognizes the contiguous candidate gadgets as the following steps. First, when we find that the address poped up by "ret" instruction is in libc, DROP records the size of the candidate gadget ending in the "ret" instruction just recognized, and initiates the variable *G_size* as 0. Then DROP increases the *G_size* when executes one instruction, until encounters the next "ret" instruction. We select the candidate gadgets with the size *G_size* no more than *T0*. And then among these gadgets, we record the length of contiguous candidate gadget sequence and choose the maximum length of the contiguous candidate gadget sequence. If the maximum length is no less than *T1*, we raise an alarm that the program contains ROP malicious code.

5 Evaluation

In this section, we choosed a large number of normal programs and ROP malicious shellcode to determine the thresholds of the two factors which represent the feature of ROP: *G_size* and *S_length*. Based on the two factors, we evaluated the false positives and false negatives of DROP with hundreds of applications and several kinds of shellcode. Finally we test the performance overhead of DROP. The evaluation is performed on an Intel Pentium Dual E2180 2.00GHz machine with 2GB memory and Linux kernel 2.6.15. Tested programs are compiled by gcc-4.0.3 and linked with glibc-2.3.5.

5.1 Statistical Analysis of Normal Programs and Shellcode

We choose hundreds of applications to test the feature of normal programs' execution, and the sizes of these applications range from 10K to 100M. These tested programs cover major categories of common programs such as Database, Media Player, Web Server. Table 3 lists the statistical results of fifteen programs. Note that the rest of programs we analyzed also come up to the average statistical result listed in Table 3.

Table 3. Statistical result of normal program

Software	LOC (K)	Benchmark	The number of candidate gadget							Max(**S_length**)						
			<=4 [1]	<=5	<=6	<=7	<=8	<=9	<=10	<=4 [1]	<=5	<=6	<=7	<=8	<=9	<=10
slocate-2.7	89.2	Search patterns in 87K database	7	13	17	30	40	48	56	1	2	2	2	3	3	5
bzip2-1.0.5	236.6	Uncompress the 269K file	7	12	15	26	34	41	46	1	2	2	2	2	3	3
man-1.6c	248.5	Open the message catalog for ls	5	10	16	30	43	51	60	1	2	2	3	3	3	4
gzip-1.2.4	278.2	Uncompress the 55M file	1	4	8	19	25	31	34	1	2	2	2	2	3	3
bc-1.06	375.9	Finds primes between 2 and limits	5	9	11	21	27	33	39	1	2	2	2	2	2	4
ngircd-0.8.1	445.1	Validate and display configuration	8	15	19	30	38	44	53	1	2	2	2	2	2	4
zgv-5.8	479.5	View JPG file	8	17	25	49	64	78	88	1	2	2	3	3	3	3
gocr-0.46	823.6	Process JPG file	6	12	15	27	33	39	47	1	2	2	3	3	3	3
grep-2.5.1	904.1	Find pattern in 1.9 MB file	2	7	9	19	26	35	40	1	2	2	2	2	3	3
openssh-2.2.1	976.8	Login in using user name	11	21	25	30	42	43	52	1	2	2	2	2	3	5
tar-1.15.1	1149.0	Uncompress the 13.6M file	12	18	25	42	55	65	77	1	2	2	3	3	3	5
gcc-4.2.4	4060.4	Compile 1KB source code	5	10	12	23	33	41	46	1	2	2	2	3	3	5
httpd-2.2.0	9883.7	ab	19	31	91	118	144	163	174	2	2	2	2	3	4	5
python-2.5.2	13602.9	Process python file	12	18	25	41	56	65	72	1	2	2	2	2	4	5
php-5.2.5	24462.0	Process php file	13	21	28	53	73	93	108	1	2	3	3	4	4	6
Average			8	15	23	37	54	58	66	1	2	2	2	3	3	4

[1] **G_size=4,5,6,7,8,9,10**

In Table 3, columns 4-10 represent the number of candidate gadgets and the length of candidate gadget is *G_size*, and columns 11-17 represent the maximum length of contiguous gadget sequence *Max(S_length)*, and each gadget has the *G_size* length. From columns 4-10, we can see that the average number of candidate gadgets is 15 in normal programs, and each candidate gadget contains no more than 5 instructions. This number is relatively small, by contrast, most of ROP malicious code contain more than 15 gadgets. To find the common number of instructions in shellcode, we statistically analyze 130 x86 shellcode from milw0rm [28]. Figure 4 shows the number of instructions in the 130 shellcode. We can see 83 shellcode among 130 shellcode we study contain more than 15 instructions, nearly 63.4%. We also rewrite these 130 shellcode by ROP technique, and find that 87 ROP shellcode contain more than 15 gadgets, nearly 66.9%.

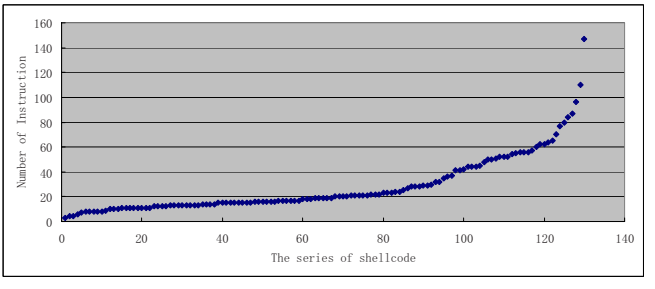

Fig. 4. The Number of Instructions in Shellcode

In addition, from columns 11-17, we can see that the larger the *G_size*, the longer the *Max(S_length)*. When *G_size* is no more than 5, the *Max(S_length)* is relatively stable and less than 2. On the contrary, based on the analysis of ROP malicious code, we find that the number of candidate gadgets is no less than 3. Malicious code uses system call to achieve malicious system operation, and the system call will be replaced by ROP technique with 3 gadgets at least. If these candidate gadgets are contiguous, the maximum length of contiguous candidate gadget sequence is more than 3. Based on the analysis of normal programs and ROP malicious code just mentioned above, we find that the threshold of *Max(S_length)* is about 3.

To further choose the thresholds of the two factors of ROP malicious code and make DROP have both low false positives and false negatives, we test a large number of normal programs and ROP malicious code which are monitored under DROP with

Table 4. The false positives and false negatives of DROP

T1 \ T0	4	5	6	7	8	9	10
1	1.000/0.000 [1]	1.000/0.000	1.000/0.000	1.000/0.000	1.000/0.000	1.000/0.000	1.000/0.000
2	0.913/0.000	1.000/0.000	1.000/0.000	1.000/0.000	1.000/0.000	1.000/0.000	1.000/0.000
3	0.000/0.009	**0.000/0.000**	0.067/0.000	0.333/0.000	0.578/0.000	0.817/0.000	0.982/0.000
4	0.000/0.069	0.000/0.023	0.047/0.023	0.053/0.023	0.073/0.023	0.235/0.023	0.637/0.023
5	0.000/0.092	0.000/0.031	0.000/0.031	0.000/0.031	0.002/0.031	0.096/0.031	0.467/0.031
6	0.000/0.104	0.000/0.054	0.000/0.054	0.000/0.054	0.000/0.054	0.000/0.054	0.067/0.054

[1] False positives and false negatives of DROP are represented in the form of *x/y*, "x" represents the false positives, and "y" represents the false negatives.

the different thresholds of *G_size* and *Max(S_length)*. Table 4 shows the experimental results. And the thresholds of *G_size* and *Max(S_length)* are represented as *T0* and *T1*, respectively. We can see that when the value of *G_size* is increasing, it makes the false positives of DROP increase and false negatives decrease, on the contrary, when the value of *Max(S_length)* is increasing, it makes the opposite result.

From Table 4, we can see that the optimal thresholds of *Max(S_length)* and *G_size* are 3 and 5, respectively, because in this case, DROP has no false positives and false negatives. Note that the thresholds of *Max(S_length)* and *G_size* can be chosen by the user. In current implementation, we focus on x86 programs, and monitor the gadgets in libc. Thus, we select the thresholds of *Max(S_length)* and *G_size* as 3 and 5, respectively.

5.2 Analysis of False Positives and False Negatives

We choose 130 Linux x86 shellcode from milw0rm [28], and all these types of shellcode are rewritten by ROP to evaluate the effectiveness of DROP. Table 5 shows ten representative cases among 130 shellcode we tested. In Table 5, column 4 represents the number of instructions in original shellcode, and column 5 represents the number of gadgets in ROP shellcode we rewrote. We can see that DROP has no false negatives. Next we also measure the false positives of DROP. Note that DROP is based on the two factors which represent the feature of ROP malicious code, and the two character factors are determined by statistically analyzing hundreds of application mentioned in Section 5.1. We select additional hundreds of applications to analyze the false positives of DROP. Experimental result shows that DROP has no false positives. In addition, although so far, in practice, we have not constructed x86 ROP malicious code by using libc gadgets to circumvent DROP. In theory, however, DROP may have false negatives.

Table 5. ROP malicious code tested on DROP

Date of Shellcode	Size	Description	Instructions	Gadgets	Detected by DROP
2009-06-16	34 bytes	setreuid(),execve("/bin/sh",0,0) [11]	16	21	✓
2009-02-20	30 bytes	chmod("/etc/shadow",666) exit(0) [8]	11	8	✓
2009-02-04	34 bytes	killall5 shellcode [9]	13	15	✓
2009-01-16	30 bytes	PUSH reboot() [10]	12	8	✓
2008-11-19	86 bytes	edit /etc/sudoers for full access [7]	29	32	✓
2007-03-09	40 bytes	/sbin/iptables -F [6]	17	19	✓
2006-11-17	45 bytes	execve(rm -rf /) shellcode [3]	23	29	✓
2006-07-04	84 bytes	portbind (define your own port) [5]	47	84	✓
2006-04-03	25 bytes	execve("/bin/sh", ["/bin/sh", NULL]) [2]	11	8	✓
2006-01-21	5 bytes	normal exit w/ random return value [4]	3	3	✓

– *Multi-stage ROP malicious code*.
Multi-stage shellcode reads the second stage payload and executes it. At the end of the first stage, it will subvert the control flow to the shellcode belonged to the second stage. At this moment, if the first stage shellcode executes "ret" instruction to jump to the second stage shellcode, it will pop up the address which is not in libc. Therefore, it may break the assumption that ROP malicious code contains no less than 3 contiguous address which are poped up by "ret" instructions within libc. In addition, if the first stage payload is short (less than 3 gadgets), it may make DROP ineffective. However, in practical analysis of shellcode, we have not found this kind of shellcode, because there is almost no chance for attacker to construct the first stage shellcode with less than 3 gadgets to read the second stage payload and jump to it.

– *Mutil-source ROP malcious code*.
  Currently, we only monitor the gadgets in libc, if the ROP malicious code uses multi-source, such as the program text segment and Linux Kernel, and constructs the gadgets in interval. DROP will be blind to this kind of malicious code. In practice, it is hard to construct multi-source ROP malicious code, because it is non-trivial to simultaneously get the base address of the multi-source.

Although there are several methods which may be potentially circumventing DROP, as demonstrated, we believe our technique can be used to defend against ROP attacks. First, these attack techniques are not practical and hard to be implemented. Second, DROP is built based on the case study of normal programs and ROP malicious code, and our experimental results show that it has no false positives and negatives.

5.3 Performance Evaluation

We used the fifteen normal applications listed in Table 6 to measure the performance of our tool DROP. For each program, we tested the performance overhead when the program runs natively and under DROP.

Table 6. Performance Overhead of DROP

Prog.	LOC (K)	Benchmark	Native Run	Under DROP	Performance Overhead
slocate-2.7	89.2	Search patterns in 87K database	0.096s	0.593s	6.2X
bzip2-1.0.5	236.6	Uncompress the 269K file	3.357s	51.860s	15.4X
man-1.6c	248.5	Open the message catalog for ls	0.188s	1.234s	6.6X
gzip-1.2.4	278.2	Uncompress the 55M file	2.457s	10.839s	4.4X
bc-1.06	375.9	Finds primes between 2 and limits	0.125s	2.628s	21.0X
ngircd-0.8.1	445.1	Validate and display configuration	0.141s	0.625s	4.4X
zgv-5.8	479.5	View JPG file	0.145s	0.703s	4.8X
gocr-0.46	823.6	Process JPG file	0.136s	1.868s	13.7X
grep-2.5.1	904.1	Find pattern in 1.9 MB file	0.958s	9.753s	10.2X
openssh-2.2.1	976.8	Login in using user name	4.626s	14.803s	3.2X
tar-1.15.1	1149.0	Uncompress the 13.6M file	8.158s	15.463s	1.9X
gcc-4.2.4	4060.4	Compile 1KB source code	0.078s	0.748s	9.6X
httpd-2.2.0	9883.7	ab	1.019s	5.208s	5.1X
python-2.5.2	13602.9	Process python file	0.725s	4.188s	5.8X
php-5.2.5	24462.0	Process php file	0.612s	2.349s	3.8X
Average			1.521s	8.191s	5.3X

From Table 6, we can see the average performance slow down factor of DROP is nearly 5.3 times. With the best of our knowledge, the performance overhead of DROP is the relatively low Valgrind overhead. The performance overhead of DROP is mainly on the recognition of "ret" instruction and statistical analysis the length of contiguous gadget sequences (*S_length*). Note that we just propose the mechanism of detecting ROP malicious code, and we believe our method can be adopted by other binary dynamic instrumentation tools, such as PIN [26], and may get better performance.

6 Discussion

We implement DROP to detect ROP malicious code, and currently DROP is based on dynamic binary instrumentation tool Valgrind [30]. Different from vulnerability-based detection tools and malicious scanning tools, our tool aims at detecting ROP malicious code. DROP has following limitations:

- *Portability Limitation*. DROP only detect ROP malicious code written on x86 architecture, however, malicious code can be rewritten on other architecture by ROP technique. Thus it will be ineffective to detect ROP malicious code on other architectures. We believe that our detecting mechanisms can be deployed to other architectures, such as SPARC.
- *Detection Limitation*. There are two limitations. First, DROP detects ROP malicious code with the assumption that it contains at least three contiguous gadgets. However, some potential shellcode methods discussed in Section 5.2 may break this assumption, and make DROP not effective. Second, currently, DROP only detects the gadgets extracted from libc. However, some techniques may help attacker use other existing library/binary, such as Linux Kernel [21], to construct ROP malicious code. DROP will not be effective for this kind of ROP malicious code.

7 Related Work

7.1 Return-into-Libc Attack

ROP attack technique fits within the larger milieu of return-into-libc attack. However, there are some critical differences between ROP attack and traditional return-into-libc attacks. Traditional return-into-libc attack leverages libc functions, whereas ROP attack uses gadgets. One gadget contains no more than five instructions and it can be easily automatically extracted from the existing library/binary. Some original defense techniques against the traditional return-into-libc attack, such as Libsafe [13], will be ineffective for the ROP attacks. Besides, ROP attack can use other existing library/binary such as Linux Kernel, and makes it more challenging to detect ROP attack.

7.2 Defense Techniques Against Code Injection and Execution

$W \oplus X$ is a technique which ensures that no memory location in a process image is marked both writable ("W") and executable ("X"), typical defending tool is PAX [1]. It forbids memory pages both writable and executable. However, ROP attack does not execute the injected code, and thus cannot be detected by $W \oplus X$.

7.3 Malicious Code Scanners

Malicious Code Scanners [15,22,24,25,32,34,35,36,38,41,42,43,45] detect the context of input, and check whether there are malicious codes. Currently, several Malicious Code Scanners detect the malicious by using pattern matching. As ROP malicious code contains the address of gadgets or data, the string in malicious code is randomized, thus malicious code scanners will be ineffective for detecting ROP malicious code.

7.4 Integrity of Control Flow

Some existing tools can be deployed to prevent the control flow of program tampering. These tools monitor the sensitive control-flow objects such as return address and function pointer. There are several typical tools [12,17,18,19], and these tools may block the pre-condition of ROP attack : altering the control flow to the location ROP malicious code exists. Our tool is an alternative approach to detect ROP malicious code based on the assumption the control flow is tampered at least once.

7.5 Memory Tainting Techniques

Memory tainting is used to defend the memory maliciously read and written. This defense technique taints the memory location at bit/byte level, and detects whether the sensitive object is corrupted by outside inputs. TaintCheck [33] is a tool which can effectively detect the control-flow hijacking. Xu et al [44] proposed a dynamic taint analysis technique to check security-sensitive operations. Several tools aim at automatically detecting malicious behavior of malicious code from network using taint analysis,such as DAKADO [20], Vigilante [16] and VSEF [31]. All these tools mentioned above are effective for defending ROP attack, as they block the ROP malicious code to be injected into memory. Our tool is an alternative approach to detect ROP malicious code based on the assumption the malicious code can be successfully injected into memory.

8 Conclusion

In this paper, we have studied Return-Oriented Programming(ROP) and wrote several ROP malicious code by using this technique. In addition, we statistically analyzed a large number of normal programs and ROP malicious code, and investigated two factors that represent the feature of ROP: *G_size* and *Max(S_length)*. Based on the observation, we found that there exist thresholds of the two factors, and can be leveraged to detect ROP malicious code by separating the ROP malicious code from normal programs. Our approach monitors program execution, and checks whether the execution comes up to the feature of ROP malicious code. We have implemented our approach in a system called DROP and applied it to analyze a number of normal programs and ROP malicious code on x86 architecture. Preliminary experimental results show that our approach is highly effective and practical, and has no false positives and negatives.

Acknowledgements

This work was supported in part by grants from the Chinese National Natural Science Foundation (60773171, 90818022, and 60721002), the Chinese National 863 High-Tech Program (2007AA01Z448), the Chinese 973 Major State Basic Program(2009CB320705), and the Natural Science Foundation of Jiangsu Province(BK2007136).

References

1. The pax project (2004), `http://pax.grsecurity.net/`
2. linux/x86 execve("/bin/sh", ["/bin/sh", null]). milw0rm (2006), `http://www.milw0rm.com/shellcode/1635`
3. linux/x86 execve(rm -rf /) shellcode. milw0rm (2006), `http://www.milw0rm.com/shellcode/2801`
4. linux/x86 normal exit w/ random (so to speak) return value. milw0rm (2006), `http://www.milw0rm.com/shellcode/1435`
5. linux/x86 portbind (define your own port). milw0rm (2006), `http://www.milw0rm.com/shellcode/1979`
6. linux/x86 /sbin/iptables -f. milw0rm (2007), `http://www.milw0rm.com/shellcode/3445`

7. linux/x86 edit /etc/sudoers for full access. milw0rm (2008), http://www.milw0rm.com/shellcode/7161
8. linux/x86 chmod ("/etc/shadow",666) & exit(0). milw0rm (2009), http://www.milw0rm.com/shellcode/8081
9. linux/x86 killall5 shellcode. milw0rm (2009), http://www.milw0rm.com/shellcode/8972
10. linux/x86 push reboot(). milw0rm (2009), http://www.milw0rm.com/shellcode/7808
11. linux/x86 setreuid(geteuid(),geteuid()),execve("/bin/sh",0,0). milw0rm (2009), http://www.milw0rm.com/shellcode/8972
12. Abadi, M., Budiu, M., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security(CCS), pp. 340–353. ACM Press, New York (2005)
13. Baratloo, A., Singh, N., Tsai, T.: Transparent run-time defense against stack smashing attacks. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, p. 21. USENIX Association, Berkeley (2000)
14. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to risc. In: Proceedings of the 15th ACM Conference on Computer and Communications Security(CCS), pp. 27–38. ACM, New York (2008)
15. Cavallaro, L., Lanzi, A., Mayer, L., Monga, M.: Lisabeth: automated content-based signature generator for zero-day polymorphic worms. In: Proceedings of the 4th International Workshop on Software Engineering for Secure Systems(SESS), pp. 41–48. ACM, New York (2008)
16. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worm epidemics. ACM Transactions on Computer Systems (TOCS) 26(4), 1–68 (2008)
17. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th Conference on USENIX Security Symposium, p. 5. USENIX Association, Berkeley (1998)
18. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., Lokier, J.: Formatguard: Automatic protection from printf format string vulnerabilities. In: Proceedings of the 10th conference on USENIX Security Symposium, p. 2003 (2000)
19. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: Pointguardtm: protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12th Conference on USENIX Security Symposium, p. 7. USENIX Association, Berkeley (2003)
20. Crandall, J.R., Su, Z., Wu, S.F., Chong, F.T.: On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proceedings of the 12th ACM Conference on Computer and Communications Security(CCS), pp. 235–248 (2005)
21. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: Proceedings of 18th USENIX Security Symposium (2009)
22. Kim, H.A., Karp, B.: Autograph: toward automated, distributed worm signature detection. In: Proceedings of the 13th Conference on USENIX Security Symposium, p. 19. USENIX Association, Berkeley (2004)
23. Krahmer, S.: X86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. Phrack Magazine (2005), http://www.suse.de/krahmer/no-nx.pdf
24. Kreibich, C., Crowcroft, J.: Honeycomb: creating intrusion detection signatures using honeypots. ACM SIGCOMM Computer Communication Review 34(1), 51–56 (2004)
25. Li, Z., Sanghi, M., Chen, Y., Kao, M.Y., Chavez, B.: Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 32–47 (2006)

26. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200. ACM, New York (2005)
27. McDonald, J.: Defeating solaris/sparc non-executable stack protection. Bugtraq (1999)
28. milw0rm: `http://www.milw0rm.com/shellcode/linux/x86`
29. Nergal: The advanced return-into-lib(c) exploits (pax case study). Phrack Magazine (2001), `http://www.phrack.org/archives/58/p58-0x04`
30. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 PLDI Conference, vol. 42(6), pp. 89–100 (2007)
31. Newsome, J., Brumley, D., Song, D.: Vulnerability-specific execution filtering for exploit prevention on commodity software. In: Proceedings of the 13th Annual Network and Distributed System Security Symposium, NDSS (2006)
32. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 226–241 (2005)
33. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software (2005)
34. Paxson, V.: Bro: a system for detecting network intruders in real-time. In: Proceedings of the 7th Conference on USENIX Security Symposium, Berkeley, CA, USA, p. 3 (1998)
35. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Network-level polymorphic shellcode detection using emulation. In: Büschkes, R., Laskov, P. (eds.) DIMVA 2006. LNCS, vol. 4064, pp. 54–73. Springer, Heidelberg (2006)
36. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Emulation-based detection of non-self-contained polymorphic shellcode. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 87–106. Springer, Heidelberg (2007)
37. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications (2009) (in review)
38. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration, pp. 229–238. USENIX Association, Berkeley (1999)
39. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS), pp. 159–169 (2004)
40. Shacham, H.: 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), pp. 552–561. ACM, New York (2007)
41. Shimamura, M., Kono, K.: Yataglass: Network-level code emulation for analyzing memory-scanning attacks. In: Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 68–87 (2009)
42. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation(OSDI), p. 4. USENIX Association, Berkeley (2004)
43. Wang, X., Pan, C.C., Liu, P., Zhu, S.: Sigfree: A signature-free buffer overflow attack blocker. IEEE Transactions on Dependable and Secure Computing 99(2) (2006)
44. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In: Proceedings of the 15th Conference on USENIX Security Symposium (USENIX-SS 2006). USENIX Association, Berkeley (2006)
45. Zhang, Q., Reeves, D.S., Ning, P., Iyer, S.P.: Analyzing network traffic to detect self-decrypting exploit code. In: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security, pp. 4–12. ACM, New York (2007)