Return-Oriented Rootkit without Returns (on the x86)

Ping Chen, Xiao Xing, Bing Mao, and Li Xie

State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University, Nanjing 210093
{chenping,xingxiao}@sns.nju.edu.cn, {maobing,xieli}@nju.edu.cn

Abstract. Return Oriented Programming(ROP) is a new technique which can be leveraged to construct a rootkit by reusing the existing code within the kernel. Such ROP rootkit can be designed to evade existing kernel integrity protection mechanism. In this paper, we show that, it is also possible to mount a new type of return-oriented programming rootkit without using any return instructions on x86 platform. Our new attack makes use of certain instruction sequences ending in `jmp` instead of `ret`; we show that these sequences occur with sufficient frequency in OS kernel, thereby enabling to construct arbitrary x86 behaviors. Since it does not make use of return instructions, our new attack has negative implications for existing defense methods against traditional ROP attack. Further, we present a design of memory layout arrangement technique for this type of ROP rootkit, whose size is not limited by the kernel stack. Finally, we propose the implementation of this practical attack to demonstrate the feasibility and effectiveness of our approach.

1 Introduction

Return-oriented programming was introduced by Shacham [31] in 2007 for the x86 architecture. It was later proved to be available on other architectures[1, 5, 7, 14, 19]. ROP allows to launch an attack by using short instruction sequences in existing libraries/executables, without injecting new code into the memory. Traditionally, the instruction sequences are chosen so that each ends in a `ret` instruction, which, if the attacker has control of the stack, transfers the flow from one sequence to the next. Based on the return-oriented programming techniques, Hund et al. [18] proposes the ROP rootkit, which leverages the existing code in Windows kernel, as such it can circumvent the kernel integrity protection mechanisms, for example, NICKLE[29] and SecVisor[30].

However, the instruction stream executed during a return-oriented attack as described above is different from the instruction stream executed by legitimate programs: first, it uses many return instructions in the instruction streams; second, it executes `ret` instructions but with no corresponding `call`; third, the ROP programs are totally installed on the stack, based on which they control the flow. When constructing the ROP rootkit, it is limited by the size of kernel stack. There are three mechanisms proposed by researchers for detecting and defeating return-oriented attacks.

The first method suggests a defense that looks for instruction streams with frequent returns. Techniques presented by Davi et al. [11] and Chen et al. [8] detect ROP based on the assumption that ROP leverages the gadget which contains no more than 5 instructions, and the number of contiguous gadgets is no less than 3. The second approach proposes a defense which is based on the fact that return-oriented instructions produce

M. Soriano, S. Qing, and J. López (Eds.): ICICS 2010, LNCS 6476, pp. 340–354, 2010.

an imbalance in the ratio of executed `call` and `ret` instructions on x86. Francillon et al. [13] proposes a hardware-based method by using a return-address shadow stack to detect ROP. With the same idea, ROPdefender[12] alternatively uses a software-based method. The third mechanism proposes a return less kernel. Most recently, Li et al. [21] propose a compiler based approach, which eliminates the `ret` instruction during program compilation.

All the current ROP defending approaches are based on the assumption that return-oriented programming uses the gadgets ending in `ret`. In this paper, we show that, on the x86, it is possible to perform return-oriented programming rootkit with the instruction sequences ending in `jmp`. For certain classes of memory errors, it is possible for an attacker to take over the kernel's control flow without executing even one return.

Our attack can circumvent existing defenses against ROP that regard the instruction snippet ending in `ret` as the property of ROP. In addition, because it is possible to launch an attack without a return, defenses that monitor the imbalance in the ratio of executed `call` and `ret` instructions will also not detect the attacks. Moreover, the return-less kernel does not eliminate the gadget ending in `jmp` instruction, which can be leveraged by us to construct the ROP rootkit without returns.

Our work makes three major contributions:

- We propose return-oriented programming rootkit without returns: rather than chaining ROP gadgets together by the `ret` instruction, control is alternatively passed to the next gadget by the `jmp` instruction. Unlike the original ROP proposals, this method avoids executing " `rets`" that are not matched with "`calls`", thus circumvents IDSes that rely on this behavior.
- We search in the binary code of linux-2.6.15, and extract the gadgets ending in "`jmp`". The gadget set can be used to do anything possible with x86 code, and by referencing the Turing-complete Language Brainfuck[25], we show that the return oriented programming without return is Turing complete.
- We construct ROP rootkit by using the gadgets without returns, and the rootkit can be leveraged to bypass not only most sophisticated kernel integrity checking mechanisms (e.g., NICKLE[29] or SecVisor[30]), but also the recently proposed ROP defense mechanisms (e.g., return-less kernel[21]).

2 Return Oriented Programming without Returns

2.1 The Frequency of Useful Instructions

Return-oriented programming traditionally uses the gadgets ending in `ret` and the function of `ret` is transferring the control flow to next gadget. In order to use the gadget without returns, we can either use the indirect `jmp` or `call` instruction. However, indirect `call` instruction suffers from the same limitation of `ret`, because existing ROP detection methods can be changed a little bit to detect the `call` without corresponding `ret`. In this paper, we leverage the gadgets ending in `jmp` to construct ROP rootkits. In x86 ISA, `jmp` instructions are interpreted as the bytes which begin with "0xff". We statistically count the number of "0xff" and "0xc3"(`ret`) in linux-2.6.15, which is 3818891 bytes. We find that there are 150643 occurrences of "0xff", accounting for

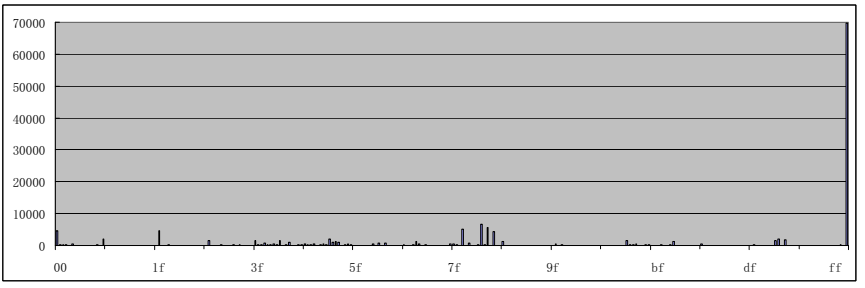

Fig. 1. Distribution of the byte after 0xff in linux-2.6.15

3.9%, compared to the 19739 occurrences of "0xc3", accounting for 0.5%. Figure 1 shows the distribution of the byte after "0xff". We can see that the most frequent occurrence is the byte "0xff"(69617). Considering the bytes after "0xff", there are 32 cases of near indirect jumps[9]: 0x20-27, 0x60-67, 0xa0-a7, 0xe0-e7. "`jmp [ecx]`"(0xff21) is the most frequent indirect jump. The near indirect jumps, which we can use to replace of `ret` instruction, is about 7115 occurrences, accounting for 0.19%. Since the far indirect jumps is determined by a 32-bit address together with a 16-bit segment selector, suppose we provide an inappropriate choice of segment selector, it will cause an exception. In this paper, we use the near indirect jump to chain the gadgets, and show that it is sufficient to construct the ROP rootkit.

2.2 Virtual *PC*

In traditional ROP, `ret` plays the important role of the "Virtual *PC*". It fetches "*PC*" value from the top of the stack. This is useful for chaining return-oriented instruction sequences because address of the gadget can be written to the stack; when the gadget is executed, reaching the `ret` instruction, then `ret` causes the next instruction sequence to be executed. Whereas in the new ROP without returns, `jmp` plays as the "Virtual *PC*". Different from `ret` instruction, `jmp` should use the register to set the "*pc*" value. To chain the gadgets together with the `jmp` instruction, we can use load, arithmetic or logic methods to set the jump register.

3 A Gadget Catalog

We use the instruction sequences ending in "`jmp * x`" to construct 30 general purpose gadgets: load immediate, move, load, store, add, add immediate, subtract, negate, not, and, and immediate, or, or immediate, xor, xor immediate, complement, shift, rotate, branch unconditional, branch conditional, finite loop and function call. The sequences were chosen automatically out of a collection of potential instruction sequences in linux-2.6.15 based on the algorithm similar with Shacham[31].

3.1 Load/Store

We consider five cases: loading a constant into a register; loading the contents of a memory location into a register; writing the contents of a register into a memory location; memory to memory data movement; writing constant 0 into a memory location.

- ***Loading a constant into a register***
  We can load six registers (`eax`,`ebx`,`esi`,`edi`,`ebp`,`ecx`) with the constant by using the following two constant-load gadgets (Gadget-1 and Gadget-2). Note that one of the important role of the constant-load gadgets is to set the registers for `jmp` instruction, as such, we can control all the gadgets to jump to the next gadget.

```
pop eax  pop ebx  pop esi  pop edi  pop ebp  jmp ecx          (1)
pop ecx  jmp edi                                             (2)
```

- ***Loading the contents of a memory location into a register***
  We choose Gadget-3 to load the value from the memory to `edx`.

```
mov edx, [eax+5]   add [eax],al  pop eax   pop ebx
pop esi  pop edi   pop ebp   jmp ecx                         (3)
```

  If we want to load the contents of a memory location into other register, we first load the value into `edx` by using Gadget-3. And the value can be stored on the stack by Gadget-4, then we adjust `esp` and get the value from the stack to register by using the constant-load gadget.

- ***Writing the contents of a register into a memory location***
  We use Gadget-4 to store `edx` into memory. Similarly, we also find the gadgets which write the contents of other registers(`eax`,`ebx`,`ecx`,`esi`,`edi`,`ebp`) to memory.

```
mov [ecx], edx  pop ebx  jmp eax                             (4)
```

- ***Memory to memory data movement***
  Since we have found the memory-load gadget (Gadget-3) and memory-store gadget (Gadget-4) which both use `edx`, we combine Gadget-3 and Gadget-4 to construct the memory-to-memory operations.

- ***Writing constant 0 into a memory location***
  Sometimes, we need to insert the zero bytes into memory, because we should load the ROP rootkit without the zero bytes. For example, when we set certain argument of function as NULL; or when we set the end of the string as NULL. At this moment, we select Gadget-5.

```
mov [edx+8],0  mov [esp+4],edx  mov ecx,[edx+40]  jmp ecx    (5)
```

3.2 Arithmetic and Logic

For all operations, we load the arithmetic and logic operand into the register or the memory. This approach allows us to achieve the memory-to-memory arithmetic and logic operations in a simple way: we load the operand into register if necessary by using the memory-load method; if the result is held in register, we write it to memory, using the memory-store gadget. Below, we describe some of the operations in details, they play as a basis for other computation behavior(e.g., control flow, function call).

– ***Neg.*** Gadget-6 can be used to compute the -x given x. Combined with the Gadget-7, we can select or ignore certain data (e.g., EFLAGS) depending on a bool value. First, we negate the bool value, then the result (-1 or 0) is do the operation "and" with the data, so that we get the data or NULL. This function is very important in the control flow gadget, the "esp offset" can be chosen by negating the value of flag.

```
neg ebx   mov [ecx+A0], ebx   btr dword ptr [ecx+424],5
mov eax,[ecx+10C]  mov [esp+8],ecx  pop ebx  jmp eax          (6)
```

– ***And.*** The wordwise "AND" gadget (Gadget-7) is useful in practice, especially when we select certain 32-bits value(e.g.,EFLAGS). If we select the value, we can "AND" this value with -1, and if we ignore it, we just "AND" this value with 0.

```
and ebp,ebx   jmp dword ptr [ecx+E985698]                     (7)
```

3.3 Control Flow

In a normal program, there are two ways to perform a branch. The branch can be an absolute address or an address relative to the current instruction. In ROP without returns, we use esp to control the target value of jump register.

– ***Unconditional Branch.*** Since in return-oriented programming the stack pointer esp takes the place of the instruction pointer in controlling the flow of execution, an unconditional jump requires simply changing the value of esp to point to a new place. This is quite easy to do by using Gadget-8.

```
pop esp  and al, 24  jmp dword ptr [ebx*4+C02E29AC]           (8)
```

– ***Conditional Branch.*** Conditional Branch uses the flags in a register called *EFLAGS* to control the direction of the control flow. We tackle the following steps in turn:

```
pushfd  mov dx,gs     jmp [ecx+C0325698]                      (9)
```

  - Use the arithmetic or logical gadget to set the flags.
  - Use Gadget-9 to store *EFLAGS* onto the stack.
  - Load the value of *EFLAGS* from the stack, and extract the specific flag, then store the specific flag on other memory location. To achieve this goal, we can choose the load/store and arithmetic/logic gadget.
  - Use the flag of interest to perturb esp conditionally. We first leverage Gadget-6 to negate the value of the memory, which contains the flag of interest, as -1 or 0. Next we use the Gadget-7 to set the "esp offset" as 0 or original value depending on the specific flag. Then we store the "esp offset" at the memory [ecx+C03E9984] which originally holds the CF flag. Finally we change the value of esp by Gadget-11. Note that, sbb will subtract the carry (CF) flag. Before executing this gadget, we simply use the Gadget-10 to clear the CF.

```
clc     jmp dword ptr [ebx+C04005E0]                          (10)
sbb esp, dword ptr [ecx+C03E9984]  jmp dword ptr [eax*4+82E0DF4]   (11)
```

– ***Finite Loop.*** Based on the gadgets mentioned above, we can achieve the finite loop with the loop number *count*. First, we use the memory-store gadget to put *count* into memory. Then we subtract it with 1, if the ZF is set, it indicates that *count* equals to 0; otherwise, it indicates *count* is larger than 0. Then we control the flow based on the value of ZF by using the conditional flow gadget. At the end of each loop, we unconditionally jump to the beginning of the loop marked as *loop start*.

3.4 Function Calls

Since the x86 instruction set architecture specifies that registers `eax`, `ecx`, and `edx` are caller-saved while registers `ebx`, `ebp`, `esi` and `edi` are callee-saved [2], we must make sure that after the function returns, the gadgets are still within the control. With this purpose, we choose Gadget-12.

```
call dword ptr [ebp-18]     jmp dword ptr [edi]                    (12)
```

3.5 Turing Complete

The gadgets without returns, which are illustrated in this section, are as powerful as Hovav's gadgets[31]. We construct gadgets to do load/store, arithmetic/logic, control flow, finite loop and function call. This set of gadgets is minimal in the sense that we can construct any program. It is an open issue that how to prove ROP language is Turing complete: being able to compute every Turing-computable function on Turing Machine[33]. One convenient and effective method is to use ROP emulate another Turing complete system. Based on the gadget set, we find that the power of our return oriented programming without returns is equal to the Brainfuck language [25], which is Turing complete. Brainfuck language is the gotoless programming language, and it has a implicit byte pointer which is free to move around within an array of 30000 bytes. In addition, Brainfuck defines only eight-instructions associated with the pointer (pointer increment/decrement;the byte at the pointer increment/decrement; read/write the byte at the pointer; start loop and end loop). When we design the ROP without returns, we define an array which can be accessed freely as well as the pointer which is used to access the array. All the behaviors of the eight instructions in Brainfuck can be done by using our gadget set. (1)pointer increment/decrement: use the "ADD/SUB" gadget. (2)the byte at the pointer increment/decrement: first use the memory-load gadget to fetch the address from the pointer, then increment/decrement the contents in the memory by "ADD/SUB" gadget.(3)read/write the byte at the pointer: use the memory-load gadget to get the address from the pointer, then load/store the value from/into the memory.(4)start loop and end loop: first read the byte at the pointer as "count", then using the loop gadget. As such, ROP without returns can emulate the Brainfuck.

4 Return-Oriented Programming Rootkit

To demonstrate that the return-oriented rootkit without returns is feasible in real system, we find a set of gadgets in the linux-2.6.15, which is 3,818,891 bytes. Each of the gadgets performs a discrete function and can be leveraged to do arbitrary computation. In this section, we would like to illustrate the design and implementation of ROP rootkit without returns.

4.1 Gadget Arrangement

As we know that, the gadgets ending in indirect `jmp` instructions use the registers, which get the data from stack memory or other part of the kernel memory. If the data is

from the stack memory, we can fetch it simply by constant-load gadget. Whereas if the data is from other kernel memory space, we need to use the memory-load gadget. In the later case, in order to get the address of the memory, we use "*control register*" in the ROP rootkit, and it can be set by constant-load gadget, or "*control register gadget*". We define the "*control register gadget*" as those gadgets which specifically set the "*control register*". The gadget `mov eax,[esp+4];mov ecx,[eax+50];jmp ecx;` is the "control register gadget", it fetches the jump target from the memory `[eax+50]` based on the "*control register*" —- `eax`, whose value is set by the memory `[esp+4]`. Note that the "*control register*" can be shared with other gadget to "memory-load" the `jmp` address or other immediate data. In current implementation, we set the value of "*control register*" on stack, and the memory, which is accessed by the "*control register*", is located on other part of the kernel space. Another problem is that where should we put the ROP rootkit image. As we know that the limited size of linux kernel stack(default 4kb or 8kb[4]) plays an important restriction for us to load the ROP rootkit. For the ROP rootkit data whose size is larger than kernel stack, we distinguish the *stack data* from other memory data. Note that the *stack data* is those data used by the stack operation such as "`pop`", "`mov reg,[esp+offset]`" and so on. The size of stack data is determined by the stack-related instructions in gadgets and the gadget arrangement. In order to cut the size of ROP rootkit on stack, we perform the following gadget arrangement strategies. (1)select the gadgets which contain less stack operation. (2) put the data which is accessed by "*control register*" (e.g., gadget address, constant value) into other kernel space. (3) put the *temp data* into other kernel space. The *temp data* can be return value of the functions, pointer value, temporary variable, object address and so on. The stack data contains the "*control register*" and half of the gadget addresses.

4.2 Experimental Step

When we finished the gadget arrangement of the return-oriented rootkit, we should load the image into the kernel memory. We can leverage the stack overflow vulnerability in linux kernel or the modules (e.g.,[35]) to load the ROP rootkit into kernel stack, and craft the control flow to the first gadget. In fact, the NIST National Vulnerability Database shows that the Linux Kernel and Windows XP SP2 have 81 and 31 buffer overflow vulnerabilities in 2006, and Sinan [27] proposes the method for smashing the kernel stack. In our experiment, by leveraging the crafted stack overflow in our dedicated module, we overwrite the return address with the first gadget's address, and next to the return address is the stack data of ROP rootkit. When executing the first gadget, `esp` points to the ROP code, and it will consecutively execute the next gadget. Figure 2 shows the experimental step of our ROP rootkit. ① represents that we leverage the buffer overflow to load the ROP rootkit into the kernel space. ② represents that we transfer the control flow to the first gadget. Then the gadgets are chained together and executed one after one(③).

Note that because we use the gadgets without returns, in practice, we can insert the ROP rootkit in other kernel space (e.g.,data segment). By contrast, the gadget with returns should rely on the stack to get the gadget address. For this consideration, we divide the ROP rootkit without returns into two parts, one part contains the data on the stack used by the stack operation such as "pop/push"; the other part (such as the address

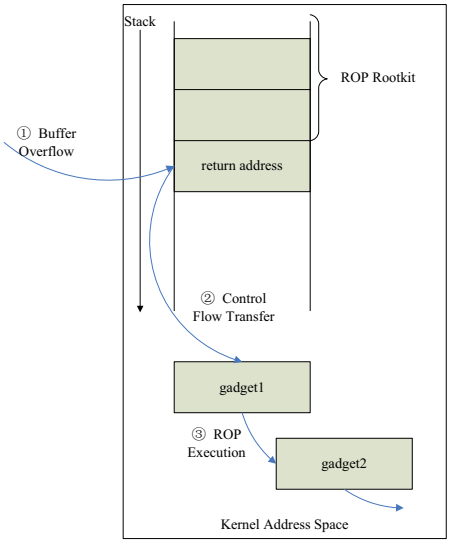

Fig. 2. Launch a ROP rootkit without returns

and data of gadget) can be located on other kernel space. To insert the other part of the ROP rootkit, we can leverage the loadable kernel modules(LKM). First in a module, we defined a buffer by "EXPORT_SYMBOL()" to ensure it will be global in scope. The buffer contains the other part of ROP rootkit data. Next, we evoke the system call "execve" to execute the "insmod" to install the loadable module with the global buffer. In experiment, we construct the ROP shellcode without returns based on the gadgets extracted from libc-2.3.5 and libgcj.so.5.0.0. Because this module contains no malicious code and does not modify the control flow, it will circumvent the state-of-the-art rootkit defenses(e.g.,NICKLE [29],SBCFI [28]). We can get the base address of the global buffer by querying "/proc/kallsyms". In addition, we leverage the kernel vulnerability to insert the stack data of ROP rootkit into kernel space, and drive the kernel control flow to the first gadget. Then it starts our ROP rootkit. The method mentioned above is practical and feasible, and it can be benefit when the size of the ROP rootkit is larger than the kernel stack size.

5 Rootkit Implementation

In experiment, we write one ROP rootkits without returns, "hide process", to demonstrate the practical feasibility of the ROP rootkit without returns. Through linux kernel's internal process list(task_struct), "hide process" searches for the process that should be hidden, then it modifies the pointers in the doubly-linked list to release the process' node. Since the operating system holds a separate, independent scheduling list, the process will still be running in the system, but not being visible by "ps" command.

Figure 3 is the source code of the rootkit, as well as its x86 instructions. We can see that there are two parts of the rootkit. The first part is to find the address of the

```
Rootkit Name : hideprocess.

Source Code:
    task = find_task_by_pid(current->pid);
    remove_parent(task);


X86 Code:
    push   pid;
    push   $0x0;
    call   find_task_by_pid;
    lea    0x60(eax),ebx;
    mov    0x60(eax),ecx;
    mov    0x64(eax),edx;
    mov    edx,0x4(ecx);
    mov    ecx,(edx);
    mov    ebx,0x60(eax);
    mov    ebx,0x64(eax);
```

Fig. 3. Source code and x86 code of hideprocess rootkit

`task_struct` by the process's pid. It leverages the function `find_task_by_pid` to achieve. Then the rootkit removes the node which represents the process. It involves three load operations and four store operations. Next, we would like to illustrate the gadget arrangement to construct ROP rootkit without returns.

5.1 Gadget Arrangement

In experiment, we achieve the "*hide process*" rootkit in the following setps.

Firstly, we leverage the gadgets sequence which is illustrated in Figure 2 to finish the function call of `find_task_by_pid`. And we get the address of `task_struct` in `eax`.

Secondly, we use the store-gadget "`xchg eax, edx; pop es; jmp esi`" to exchange `eax` to `edx`, and use Gadget-4 to save the return value on the stack. Next, based on the load and store gadget as well as the gadget "`add [edx],eax; adc [esi],al; jmp [edi]`", we calculate four memory addresses (marked as ①, ②, ③, and ④ in Figure 4): `eax+0x60`, `eax+0x64`, `[eax+0x60]+0x4`, and `[eax+0x64]`, and store them in the memory.

Thirdly, we use four memory-to-memory store gadgets to modify the contents in the destination memory `[[eax+0x64]]` ([①] → [④] in Figure 4), `[[eax+0x60]+0x4]` ([②] → [③] in Figure 4), `[eax+0x60]` (① → [①] in Figure 4) and `[eax+0x64]` (① → [②] in Figure 4), as such, we modify the doubly-linked list to hide the process. Once the process hiding is finished, the rootkit performs a transition back to the vulnerable code to continue the normal execution.

Figure 4 shows the gadgets which can achieve the "hide process" rootkit. Following the steps mentioned above, we select the 101 gadgets to construct the ROP rootkit. (1) We leverage the function call gadgets (gadget 1-12 in Figure 4) to invoke the function `find_task_struct`. (2) We store the return value of `find_task_struct` into the memory (gadget 13-19 in Figure 4). (3) Because the rootkit needs to use the constant

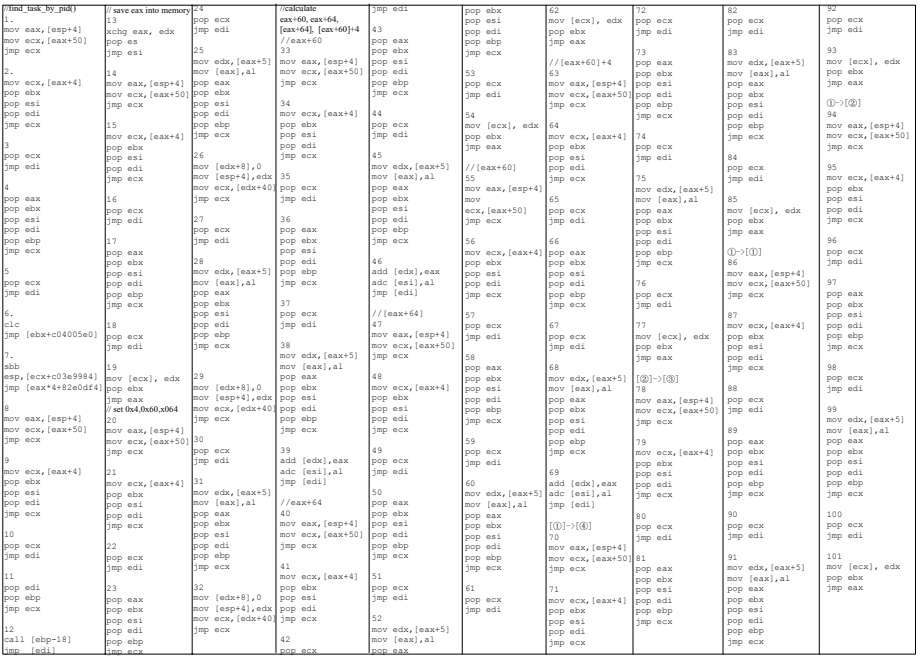

Fig. 4. Gadgets selected for hide process

value *0x4*, *0x60*, and *0x64* to calculate the memory address, we select Gadget-5 in Section 3.1 to insert the zero bytes in the memory (gadget 20-32 in Figure 4). (4) Based on the return value obtained in the second step, we calculate the the link nodes' addresses, and store them into the memory (gadget 33-69 in Figure 4). In this step, we use Gadget-3 in Section 3.1 to load the return value and the gadget "`add [edx],eax; adc [esi],al; jmp [edi]`" to calculate the address `eax+0x60` and `eax+0x64`, then we use the Gadget-4 in Section 3.1 to store the memory addresses into the memory (gadget 33-46 in Figure 4). We use the memory-to-memory data movement gadgets to store the address `[eax+0x60]` and `[eax+0x64]` into the memory (gadget 47-62 in Figure 4). Based on the address `[eax+0x60]`, we calculate the address `[eax+0x60]+0x4` and store it into the memory (gadget 63-69 in Figure 4). (5) We combine the Gadget-3 and Gadget-4 to change the process links (gadget 70-101 in Figure 4).

In experiment, the total size of the ROP rootkit "hide process" is 1812 bytes, and the stack data is about 876 bytes, which occupies 48.3%. That is to say, we can move more than half of the data from the stack to other kernel space. Note that because the size of "hide process" is no larger than 4kb, we can also put it totally on the kernel stack.

We list the snippet of our ROP rootkit "hide process" as following. It is divided into two parts: one part is on the stack which begins from the stack address "0xcabe7de0". The other part is global buffer which begins at "0xe0822440".

The snippet of the stack data is as belows:

```
90909090 3c2482c0 203116c0 aad634c0 90909090 0c1f4220
90909090 19961ec0 90909090 0c8b4320 90909090 902482e0
140a35c0 ...
```

The snippet of the global buffer data is as belows:

```
743a1cc0 90909090 90909090 90909090 90909090 90909090
90909090 90909090 90909090 90909090 90909090 90909090
90909090 90909090 90909090 90909090 90909090 90909090
90909090 74b311c0 d8ffffff 0cda12c0 90909090 ...
```

Besides process hiding, arbitrary rootkits, that contain no persistent code to be called on demand[28], can be implemented in the same way: ROP rootkit needs to exploit the vulnerability in order to gain control and ROP rootkit can be divided into two parts, one is on the stack, the other is loaded by the "insmod" in other kernel space.

We would like to mention that the rootkit which contains the persistent code to be called on demand is much more difficult to be implemented. Take "taskigt" for example, it adds hooks to the /proc filesystem that cause their functions to be executed when a user program reads from certain /proc entries. It provides a hook that grants the root user and group identity to any process that reads a particular /proc entry. The code of the "taskigt" can be divided into two parts, one is to create a particular /proc entry and set the read routine for it. The other part is the callback routine which is used for granting the root identity to the process that reads the particular "/proc" entry. In practice, we have rewritten the former part of "taskgit" with our ROP rootkit without returns, which is 283 bytes, and the stack data is about 103 bytes(occupies 36.4%). But currently, we have not achieved another part of the rootkit "taskigt", the persistent callback routine that grants the root identity, by using ROP techniques. This is because that the function pointer "read_proc" in struct "proc_dir_entry" should point to the callback routine. And we can not load the ROP callback routine totally in a kernel space with a fixed memory location, and part of it must be on the kernel stack. Therefore, the ROP callback routine may be overwritten or out of the control during kernel running. However, if the gadgets do not use the stack operation so that ROP callback routine can be loaded in a fixed memory location, we may achieve the persistent callback routines by ROP techniques.

6 Discussion

In this section, we discuss several issues related to the ROP rootkit without returns. First, in current implementation, the ROP rootkit is constructed manually. Further, we can design a ROP language and implement a compiler which acts as an abstraction of the concrete gadget sets so that developers do not have to focus on the intricacies of the arrangement of the gadgets on the lowest layer. Second, compared with the ROP rootkit using returns, our rootkit should be more carefully designed, because indirect jmp instruction uses the register whose value should be set by other gadgets. For this reason, we may frequently use the constant-load gadget or the *control register gadget* to set the value of the register. Third, in current implementation, we put the part of

the ROP rootkit on the kernel stack because some of the gadgets have the stack related operations. However, `jmp` needs not fetch the target address from the stack, if we find the gadgets which do not leverage the stack operation and through the elaborately construction, we can install the ROP rootkit totally out of the kernel stack.

7 Related Work

7.1 Return Oriented Programming

Return-oriented programming is developed from the return-into-libc attack techniques [20, 22, 26]. The common feature of them is borrowing the code from existing code. Different from the return-into-libc attack, ROP leverages the short instruction sequence rather than functions. Schacham [31] has proposed the ROP attack in 2007, and he also shows that ROP which leverages the gadget ending in `ret` instruction is Turing complete on x86 architecture. Further, several researchers found that ROP can be available on other architectures [1, 5, 7, 14, 19]. However, all the ROP techniques mentioned above use the gadget ending in "ret" instruction, they suffer from the current ROP defenses [8, 11–13, 15, 21].

There are two other parallel and independent efforts in improving existing ROP techniques [6, 34]. S. Checkoway et al. [6] proposed the method which aims at constructing ROP shellcode to escape existing defenses. They use the *return like instruction sequence* (`pop-jmp`) to launch the ROP attack. Although *return like instruction sequence* can be leveraged to chain the gadgets, but it frequently uses the gadget "`pop ebx; jmp [ebx]`", whose behavior is similar with `ret`, therefore current ROP defenses based on the statistics checking techniques [8, 11] can defend it with a little modification. Also, their implementation highly relies on the caller-saved register "`edx`" to chain the instruction sequences, it is an issue to invoke the system call or function all without the specific instructions "`pop edx;jmp [edx]`". Concurrently and independently, Bletsch et al.[34] proposed the concept "Jump-Oriented Programming (JOP)", with an attack example as well. However, our techniques are quite different compared to theirs. They leverage the gadget ending in `call` instruction without corresponding `ret` instruction. Existing ROP defending tools [12] can be modified a bit to detect their JOP shellcode with independent `call` instruction by monitoring the imbalance of "call-ret" stack. In our work, we use the gadget ending in `jmp` to eliminate the independent `call` or `ret`. In this paper, we further extract the gadgets without returns in OS kernel, and demonstrate the gadgets ending in `jmp` instruction is Turing complete, we can construct the ROP rootkit without returns.

7.2 Kernel Integrity Protection Mechanisms

The main idea of the kernel integrity protection mechanisms is that the kernel memory should be protected against unauthorized injection of code. Livewire [16] aims at protecting the guest OS kernel code and critical kernel data structures from being modified based on a virtual machine monitor (VMM). However, it can not prevent the code injection by exploiting the kernel vulnerability or loadable module. To prevent the malicious code injection by loadable module, kernel module signing proposed the trusted

rootkit certification authority(CA) to check whether there is a valid digital signature in every kernel module. If this check fails, it prevents to load the code. This technique has been implemented most notably for Windows operating system since XP [23] and is used on every new Windows system. Further, NICKLE [29] is a tool which leverages the shadow memory techniques to forbid the attempt to execute unauthenticated code, as such, a kernel rootkit will never be able to execute any of its own code. $W \oplus X$ is another technique which prevents the memory from being both writable and executable, therefore, it prevents the injected code from being executed. This mechanism has been introduced into most of existing operating system, including linux kernel and Windows operating system. For example, the PaX[32] and Exec Shield patches[36] for Linux, PaX for NetBSD, Data Execution Prevention (DEP) [24] for Windows XP Service Pack 2 and Windows Server 2003. In addition, secVisor [30] is a software solution that leverages a hypervisor, which restricts what code can be executed by a modified Linux kernel,to enforce $W \oplus X$. However, as the ROP rootkit contains no malicious code, but only the existing code in linux kernel, therefore, it can circumvent the kernel integrity protection mechanisms.

7.3 Kernel-Level Control Flow Integrity

Currently, the first step of our ROP rootkit should leverage the vulnerability in linux kernel or its loadable module to maliciously transfer the control flow and execute the first gadget. There are several control flow integrity protection methods in kernel[3, 17, 28] and Pointer integrity checking method[10]. The mechanisms of kernel-level control flow integrity strategy are tend to be trade-off performance and precision[28]. Therefore, it will bring the false negative. Moreover, current OS kernel is extensible with loadable kernel modules(LKM), and some of them unavoidably contain the bugs for the attacker to leverage. As such, in kernel, it is possible for us to transfer the control flow and achieve the first step of the ROP rootkit without returns. In this paper, we focus on kernel integrity protection mechanisms and not on control-flow integrity.

8 Conclusion and Future Work

In this paper we presented the design and implementation of return-oriented rootkit without the returns. This techniques can be used to enhance existing return-oriented rootkit proposed by Hund et al. [18], which can be prevented by the return less kernel proposed by Li et al.[21]. Based on the gadgets ending in `jmp`, we can construct the ROP rootkit without returns, which can evade most of existing kernel integrity checking methods and the state-of-the-art ROP detection approaches. Moreover, Li's compiler [21] gives us the source of instruction sequences ending in jmp to construct the ROP rootkit. Thus it demands the new ROP protection mechanisms. In the future, we plan to extend the research to design the automatical framework which can be used to write the ROP rootkit without returns. And we also would like to investigate the effective detection mechanism for return-oriented rootkits without returns.

Acknowledgements

This work was supported in part by grants from the Chinese National Natural Science Foundation (60773171, 61073027, 90818022, and 60721002), the Chinese National 863 High-Tech Program (2007AA01Z448), and the Chinese 973 Major State Basic Program(2009CB320705).

References

1. Felix "fx" lidner. Developments in cisco ios forensics. CONFidence 2.0, http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf
2. The x86 instruction set architecture, http://www.ugrad.cs.ubc.ca/~cs411/2009W2/downloads/x86.pdf
3. 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, New York (2005)
4. Bovet, D.P., Cesati, M.: Understanding the linux kernel, 3rd edn., p. 85. O'Reilly Media, Inc., Sebastopol (2006)
5. 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, pp. 27–38. ACM, New York (2008)
6. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: 17th ACM Conference on Computer and Communications Security (2010)
7. Checkoway, S., Feldman, A.J., Kantor, B., Halderman, J.A., Felten, E.W., Shacham, H.: Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. In: Proceedings of EVT/WOTE 2009. USENIX/ACCURATE/IAVoSS (2009)
8. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: Drop: Detecting return-oriented programming malicious code. In: Prakash, A., Sen Gupta, I. (eds.) ICISS 2009. LNCS, vol. 5905, pp. 163–177. Springer, Heidelberg (2009)
9. Corporation, I.: Ia-32 intel architecture software developers manual. Instruction set reference, vol. 2 (2006)
10. Dalton, M., Kannan, H., Kozyrakis, C.: Real-world buffer overflow protection for userspace & kernelspace. In: Proceedings of the 17th Conference on Security Symposium, SS 2008, pp. 395–410. USENIX Association, Berkeley (2008)
11. Davi, L., Sadeghi, A.R., Winandy, M.: Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In: Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing, pp. 49–54 (2009)
12. Davi, L., Sadeghi, A.R., Winandy, M.: Ropdefender: A detection tool to defend against return-oriented programming attacks. Technical Report HGI-TR-2010-001 (2010), http://www.trust.rub.de/home/_publications/LuSaWi10/
13. Francillon, A., Perito, D., Castelluccia, C.: Defending embedded systems against control flow attacks. In: Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code, SecuCode 2009, pp. 19–26. ACM, New York (2009)
14. Francillon, A., Castelluccia, C.: Code injection attacks on harvard-architecture devices. In: Syverson, P., Jha, S. (eds.) Proceedings of CCS 2008, pp. 15–26 (2008)
15. Frantzen, M., Shuey, M.: Stackghost: Hardware facilitated stack protection. In: Proceedings of USENIX Security 2001, pp. 55–65 (2001)

16. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proc. Network and Distributed Systems Security Symposium (February 2003)
17. Grizzard, J.: Towards self-healing systems:re-establishing trust in compromised systems. In: PhD thesis. Georgia Institute of Technology (2006)
18. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: Proceedings of 18th USENIX Security Symposium, San Jose, CA, USA (2009)
19. Kornau, T.: Return oriented programming for the arm architecture. Master's thesis, Ruhr-Universitat Bochum (2010), http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf
20. 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
21. Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating return-oriented rootkits with 'return-less' kernels. In: Proceedings of the 5th ACM SIGOPS EuroSys Conference, EuroSys 2010 (2010)
22. McDonald, J.: Defeating solaris/sparc non-executable stack protection. Bugtraq (1999)
23. Microsoft: Digital signatures for kernel modules on systems running windows vista (2007), http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/kmsigning.doc
24. Microsoft: A detailed description of the data execution prevention (dep) feature in windows xp service pack 2 (2008), http://support.microsoft.com/kb/875352
25. Mueller, U.: Brainfuck: An eight-instruction turing-complete programming language, http://www.muppetlabs.com/~breadbox/bf/
26. Nergal: The advanced return-into-lib(c) exploits (pax case study). Phrack Magazine (2001), http://www.phrack.org/archives/58/p58-0x04
27. noir: Smashing the kernel stack for fun and profit. Phrack Magazine (2006), http://www.phrack.com/issues.html?issue=60&id=6
28. Petroni, N., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS), pp. 103–115. ACM, New York (2007)
29. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
30. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 335–350. ACM, New York (2007)
31. 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)
32. Team, P.: Documentation for the pax project overall description (2008), http://pax.grsecurity.net/docs/pax.txt
33. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proc. London Math. Soc., 230–265 (1936)
34. Bletsch, T., Jiang, X., Freeh, V.: Jump-oriented programming: A new class of code-reuse attack. Technical Report TR-2010-8 (2010)
35. Viro, A.: Linux kernel sendmsg() local buffer overflow vulnerability (2005), http://www.securityfocus.com/bid/14785
36. Wikipedia: Exec shield, http://en.wikipedia.org/wiki/Exec_Shield