Automatic Construction of Jump-Oriented Programming Shellcode (on the x86)

Ping Chen†, Xiao Xing†, Bing Mao†, Li Xie†, Xiaobin Shen‡, Xinchun Yin‡
†State Key Laboratory for Novel Software Technology
Dept. of Computer Science and Technology
Nanjing University, China
{chenping,xingxiao,maobing,xieli}@nju.edu.cn
‡ College of Information Engineering
Dept. of Computer Science
Yangzhou University, China
{xbshen,xcyin}@yzu.edu.cn

ABSTRACT

Return-Oriented Programming (ROP) is a technique which leverages the instruction gadgets in existing libraries/executables to construct Turing complete programs. However, ROP attack is usually composed with gadgets which are ending in `ret` instruction without the corresponding `call` instruction. Based on this fact, several defense mechanisms have been proposed to detect the ROP malicious code. To circumvent these defenses, Return-Oriented Programming without returns has been proposed recently, which uses the gadgets ending in `jmp` instruction but with much diversity. In this paper, we propose an improved ROP techniques to construct the ROP shellcode without returns. Meanwhile we implement a tool to automatically construct the real-world Return-Oriented Programming without returns shellcode, which as demonstrated in our experiment can bypass most of the existing ROP defenses.

1. INTRODUCTION

Attackers often construct malicious code to gain unauthorized control and perform malicious actions. Traditional method injects the malicious code into program's memory space by exploiting software vulnerabilities, and hijacks the control flow to execute the injected code. To prevent such code injection attacks, researchers have proposed a large number of techniques, such as vulnerability-based signature (e.g., [1]), exploit-based signature [2, 3], and non-executable memory (e.g, PAX [4]); and they all rely on the fact that malicious executable code will be received from outside. However, this hypothesis is broken by return-into-libc attack [5–7], which uses the existing library functions to construct the attack. Unfortunately, return-into-libc attack leverages the very specific library functions, and it cannot perform any arbitrary computations.

To get rid of the limitation of return-into-libc attack, Shacham [8] proposed Return-Oriented Programming (ROP), which reuses the short instruction sequences in existing libraries or executables. These instruction sequences end in `ret`, and chain with each other to construct the malicious code. ROP attack has become an actual threat in practice [9–11] and can be mounted on modern architectures [9, 10, 12–14]. There are several approaches for defeating the ROP attack, from both hardware and software perspectives. Francillon et al. [15] proposed a hardware-based approach which uses an embedded microprocessor to prevent the call/ret stack from being maliciously damaged. ROPdefender [16] is a software based approach. It uses the shadow memory to check the violation of the invariant of return address. Davi et al. [17] and Chen et al. [18] propose the techniques that detect ROP based on the assumption that ROP leverages the contiguous gadgets ending in `ret`, which contain short instructions. To eliminate the `ret` instruction, Li et al. [19] propose a compiler based approach.

Most recently, Checkoway et al. [20] proposed the advanced Return-Oriented Programming, which leverages no return instruction at all, to circumvent the ROP defenses. They demonstrate the Return-Oriented Programming without return is feasible on both x86 [21] and ARM [22]. However, the techniques proposed by Checkoway et al. [20] must be manually designed to construct the shellcode or even infeasible for constructing the sophisticated shellcode. As such the work is time-consuming and tedious. In this paper, based on more sufficient and carefully designed gadgets in the libraries, we proposed a Jump-Oriented Programming (JOP) to improve existing Return-Oriented Programming without returns techniques. Based on the new techniques, we present a tool which can automatically construct the Jump-Oriented Programming shellcode. In particular, we make the following contributions in this paper:

- We proposed an improved ROP techniques. Compared to the state-of-the-art Return-Oriented Programming without returns [20], our method is more feasible to construct the programs, without the manual tedious work. Particularly, we use the *combinational gadget* to invoke the system calls (Section 2.1) and the *control gadget* to set the jump register (Section 3.1).

- We propose techniques to automatically construct the gadgets, the basic building block in Jump-Oriented Programming, and show these gadgets are also Turing complete [23].

- We have implemented these techniques into a tool, and applied it to automatically construct a large number of real world Jump-Oriented Programming shellcode from `milw0rm` [24]. Experimental results show that our tool can efficiently construct the shellcode within few seconds.

ASIACCS '11, March 22–24, 2011, Hong Kong, China.

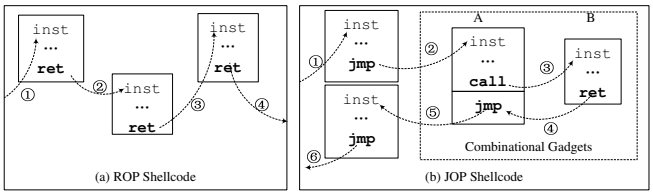

Figure 1: The structure of ROP and JOP Code

2. OVERVIEW

2.1 The Building Blocks– JOP Gadget

The structure of the traditional ROP shellcode is that the gadgets are chained together by `ret` instructions [8]. As Figure 1(a) shows, traditional ROP shellcode is quite different from the normal program: it executes several small instruction snippets with an independent `ret` instruction. Similarly (but with more challenges) as shown in Figure 1(b), we could have a "Jump-Oriented Programming(JOP)": rather than using the independent `ret` instruction to construct the gadget, control can be passed to the next gadget by the `jmp` instruction or by combinational `call` and `ret` instructions.

Compared to existing Return-Oriented Programming without returns [20], there are at least two differences. First, although JOP shares the same idea of using the gadgets which end in indirect jump, we propose the *control gadget* to specifically set the jump register, which is the key of control flow automatic transfer. We will illustrate the gadgets in Section 3. Second, we introduce a special gadget, *combinational gadget*, which ends in combinational `call` and `ret` instruction. Because certain functional code snippet ending in indirect jump (e.g., kernel trapping gadget) can not be found in x86 code. The work flow of the *combinational gadget* is that, if gadget A (Figure 1(b)) ends in the instruction sequence "`call`, `jmp`" and gadget B ends in `ret`, gadget A can call gadget B (③ in Figure 1(b)), which will return to gadget A's `jmp` instruction (④ in Figure 1(b)), and then it jumps to next gadget (⑤ in Figure 1(b)). *Combinational gadget* is useful in constructing the shellcode, because shellcode often leverages the system call to achieve its goal, and we can construct the kernel trapping gadget by using *combinational gadget* to invoke the system call (Section 3.2.5).

2.2 Where is the JOP Gadget

As JOP gadget ends with a jump instruction, we should search the libraries or executables to find out those instructions which end with a `jmp`. There are two types of jump instructions: direct jump and indirect jump. In JOP, we would like to use the indirect jump instead of direct jump, because it is often the register value (used in indirect jump) rather than constant value (direct jump) that can be controlled by the shellcode. Also, there are two types of indirect jumps: *near* jump and *far* jump. In practice, we find the gadgets ending in the *near* indirect jump are sufficient to construct the JOP shellcode. However, for *far* indirect jump, we have to make the appropriate choice of segment selector when constructing far jump code. As such, in this paper, we mainly focus on how to construct *near* indirect jump shell code.

There are 32 cases of *near* indirect jumps, which have the following hexadecimal values: 0xff 0x20-0x27, 0xff 0x60-0x67, 0xff 0xa0-0xa7, 0xff 0xe0-0xe7. Based on these hexadecimal values, we could hence select the gadgets from existing executables and libraries. Algorithm 1 shows the approach on how to find the gadgets ending in near `jmp` instruction. Note "*Gadget*" represents a JOP gadget, *"L_t"* indicates the size of the text segment of the libraries or executables, *"δ_m"* indicates the maximum length of the instruction (20 bytes in x86 [8]), and *"J_r"* is the register used by the `jmp` instruction. Given a library or executable, the searching algorithm can be divided into two procedures: the first procedure *Create_Gadget* is to search the `jmp` instruction in the whole text segment of the library or executable (line 3-4). If it finds such an instruction, it creates two sets: `Gadget` which contains the jump instruction and J_r which contains the jump registers(line 5). Then it runs into the second procedure *Build_Subgadget* (line 6). In the second procedure, it searches the valid instruction before the `jmp` instruction, and puts the instructions into the `Gadget` one by one (line 11-15). The algorithm stops searching in either of the two cases (line 13): (1) when it encounters return instruction, jump instruction and direct call instruction, note that indirect call instruction can be used to construct the *combinational gadget* ending with `call-jmp` instructions; (2) when it encounters the instruction, which contains the register of `jmp` instruction but does not use it as destination operand (jump register conflicts). Note that the suffix of *Gadget* may also be an useful instruction sequence, and it will have a different function. For example, if *Gadget* is in the form of "*a;b;jmp*", then "*b;jmp*" is also a JOP gadget.

Algorithm 1 Gadget Searching Algorithm

```
1: /* Gadget: a JOP gadget; L_t: the size of the text segment of
   the libraries or executables; δ_m: the maximum length of the in-
   struction; J_r: the register used by the jmp instruction. insn.Reg
   represents the registers used in instruction insn.*/
2: Create_Gadget() {
3:   for (pos=1; pos < L_t; pos++){
4:     if( the word at pos is indirect jump instruction){
5:         Gadget ={jmp}; J_r ={jump registers};
6:         Build_Subgadget(pos,jmp,Gadget);
7:     }
8:   }
9: }
10: Build_Subgadget(pos, pre_insn, Gadget){
11:   for (step=1; step < δ_m; step++){
12:     if (bytes [(pos-step)...(pos-1)] decode as a valid instruction
   insn){
13:       if (¬(insn == "ret"/"jmp" /"direct call") ∧¬ (insn.Reg ⊂
   J_r ∧ Reg is not set by insn) {
14:         Put insn after pre_insn in the Gadget;
15:         Build_Subgadget(pos-step,insn,Gadget);
16:       }
17:     }
18:   }
19: }
```

3. DISCOVERING TURING COMPLETE JOP GADGETS

Once we have understood the basic forms of JOP gadget, in this section we examine what kind of instruction sequences can be leveraged to construct the JOP gadget, and whether JOP gadget could have the same power as ROP in terms of Turing completeness. We use two widely used libraries, `libc-2.3.5.so` (C library) and `libgcj.so.5.0.0` (a Java runtime library), as the code base. Note that the code base can use any libraries or executable code, here we select these two representative libraries to show the feasibility of our approach.

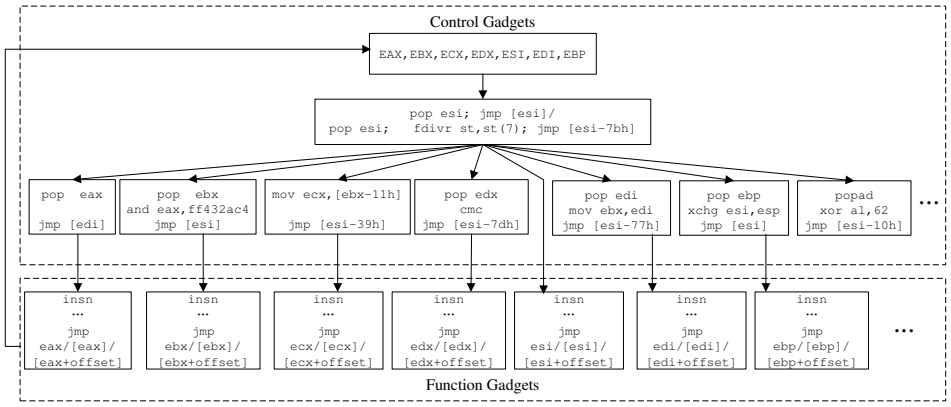

Figure 2: An Overview of the Design of Jump-Oriented Programming Gadget

3.1 Control Gadget

In JOP gadget design, we leverage the jump instruction to connect the gadgets. Different from the `ret` instruction, indirect jump instruction uses the registers (except `ESP`), which are called *jump register*. *Jump register* can be assigned as the gadget address (e.g., `jmp esi`), the memory location (e.g., `jmp [esi]`), or the memory location with an offset (e.g., `jmp [esi+offset]`). In our design, we primarily select the *control gadgets* in the form of "`pop-jmp`" or "`mov-jmp`" to set the *jump register*. Note that the "`pop/mov`" instruction and `jmp` instruction may use different registers (e.g., `pop ebp; jmp [esi]`). If this is the case, we must use more than one register to construct the *control gadget*. However, there could exist the data dependency between the *jump registers* and it may lead to the circles (e.g., `EAX`→`EBX`→`EAX` from "`pop eax; jmp [ebx]`" and "`pop ebx; jmp eax`"). The first gadget can set *jump register* `EAX`, but it needs its *jump register* `EBX` to be set earlier, whereas in the second gadget, `EAX` should be set earlier than `EBX`. In order to break the circle and control all the *jump registers*, we release `ESI` and use it specifically as *jump register* in the *control gadget*. We choose `ESI` based on the following considerations. First, `ESI` is the callee-saved register. Second, in experience of writing the shellcode in milw0rm [24], we find shellcode often uses the system call with less than four arguments, therefore `ESI` will not be used to set arguments [25]. Third, we find `ESI` occurs with the high frequency as the *jump register* in the gadgets, and other gadgets, whose *jump register* is not `ESI`, can also be indirectly set by the *control gadgets* whose *jump register* is `ESI`. To set `ESI`, we find two alternative *control gadgets*, Gadget-1 (`libgcj.so.5.0.0`) and Gadget-2 (`libc-2.3.5.so`).

```
1  pop esi        (1)      1 pop esi          (2)
2  jmp [esi]               2 fdivr st,st(7)
                           3 jmp [esi-7Bh]
```

For the other *jump registers*, we select the *control gadgets* which can set the register and use `ESI` as *jump register*. For example, Gadget-3 (shown below) is used to set `EDI`. We also find similar *control gadgets* for `EBP`, `EBX`, `ECX` and `EDX`, as shown in Figure 2. To set *jump register* `EAX`, we use "`pop eax; jmp [edi]`", in which `EDI` can be set by Gadget-3. To set the *jump registers* `ESI` in these *control gadgets*, we use Gadget-1 and Gadget-2. Combined with the gadgets mentioned above, we can control any *jump register*. But this is not an absolute requirement. *Control gadget* can also be the gadget which not only sets other register but also its own *jump register*, such as Gadget-5, which will be illustrated in section 3.2.1.

```
1 pop edi        (3)      1 pop esp          (4)
2 mov ebx, edi            2 jmp [esi]
3 jmp [esi-77h]
```

Since most of the *control gadgets* use "`pop/mov`" instructions to set the *jump register* from the stack. The gadget address should be fetched by `ESP`. In addition, several *function gadgets* use the "`pop/mov`" instruction to get the address or the data from stack. Thus, we require the gadget that can set `ESP`. To achieve this goal, we select Gadget-4 (`libc-2.3.5.so`).

3.2 Function Gadget

Function gadgets can be used to achieve data movement, data arithmetic operation, logic operation, unconditional branch, conditional branch, system call, and function call.

3.2.1 Data Movement

Load gadgets play a critical role in setting registers. "*Control gadgets*" described in Section 3.1 can also be used as *load gadgets*. In order to set a pile of registers, alternatively, we introduce Gadget-5 (`libgcj.so.5.0.0`) to set them. In practice, we also use Gadget-5 to set the *jump registers*.

```
1  popad          (5)     1  pushad          (6)
2  xor al,62              2  aam
3  jmp [esi-10h]          3  jmp [esi-70h]
```

Store gadgets are used to put the content of a register into a memory location. We find the gadgets which can store `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, and `EBP` into the memory from our code base. But no gadget can set `EAX` into the memory. Thus, to store the return value (`EAX`) of system call to memory, we use the gadgets "`xchg eax, edi; cmp ah, dh; jmp [esi-77h]`" to exchange the return value to `edi` and the *store gadget* "`mov [ebx-2],edi;jmp [esi-77h]`" to save `edi` to memory. In addition, sometimes, when invoking the function call or executing the loop body, we need to save registers (caller-saved registers or registers used in loop body) in the memory, we could select Gadget-6, which is from `libc-2.3.5.so`, to save all the registers.

The shellcode should not contain any NULL bytes, however it often needs to get the value with NULL bytes (e.g., system call index) from the memory during execution. In practice, we use Gadget-7 (`libgcj.so.5.0.0`) to compute arbi-

trary constant value and store it in the memory by adding two values without NULL bytes, which can be naturally preset in JOP shellcode. For example, if we want to set system call number (`0xb`) of `execve` in the memory, we can store the two values (e.g., `0x1111111f` and `0xeeeeeeec`) in the JOP shellcode, then we ensure one value in the memory `[ebx]` and load the other value to `edi`, finally the memory `[ebx]` will get the addition result (`0xb`).

```
1  add [ebx],edi        2  jmp ebp        (7)
```

3.2.2 Data Arithmetic and Logic Operation

For data arithmetic and logic operations, we put the corresponding operands into register or memory by *load/store* gadgets. Then we can compute any arithmetic/logic operations in a simple way: we load the operands into register as needed by using *load gadgets*; if the result is now held in register, we write it to memory, using the *store gadgets*. We searched `libc-2.3.5.so` and `libgcj.so.5.0.0` and found 115 gadgets, which achieve the arithmetic/logic operations (e.g., `add`, `sub`, `inc`, `dec`, `xor`, `and`, `or`, `not`, `neg`, `rol`).

3.2.3 Control Flow

Unconditional branch uses the Gadget-2 and Gadget-4 to direct the control to the destination gadget and change the `ESP` to make the successive gadgets fetch their data from the stack.

Conditional branch uses the flags in `EFLAGS` to drive the direction of the control flow. The strategy we developed has four steps, (1) Undertake arithmetic or logical gadget that sets (or clears) flag of interest. (2) Store `EFLAGS` on the stack by `pushfd` instruction. (3) Get the flag from the stack. To achieve this goal, we use the gadget "`or edx, ebp;jmp [eax]`", which is extracted from `libgcj.so.5.0.0`. `EBP` is set by all "1" except for the flag bit (e.g.,`ZF=0`, `EBP=0xffffffbf`), and `EDX` is set as the `EFLAGS`. If the flag of interest equals to "1", then `EDX` will get -1, on the other hand, `EDX` will be equal to `EBP`. (4) Use the flag of interest to perturb jump target conditionally. We use the gadget "`jmp [edx+esi]`", which is from `libgcj.so.5.0.0`, to change the flow. If we store the different addresses in memory `[esi-1]` and `[esi+ebp]`, `EDX+ESI` will point to the different gadgets. There is an exception that, because `CF` is the last bit in `EFLAGS`, after executing "`or edx, ebp;jmp [eax]`", the value of `EDX` will be 1 distance depending on the flag is set or cleared (`CF`=0, `EDX`=-2; `CF`=1, `EDX`=-1). In this scenario, there exists a memory usage conflicts because the address of the next gadget is 4 bytes. To solve the problem, we use "`rcl edx,cl; jmp [eax]`" to change the `EFLAGS` before executing "`or edx, ebp;jmp [eax]`".

3.2.4 Finite Loop

We can achieve the finite loop by the gadgets in Listing 1. In JOP shellcode, "*loop body*" can be a series of gadgets which achieve the specific functions such as decoding the malicious code. To be simple, we represent it as gadget 2 in Listing 1. Suppose *iterate number* " `count`" has been stored in the memory, first we regulate the `ESP` (gadget 4 in Listing 1). Then we subtract "`count`" by 1 (gadget 5, 6 in Listing 1), if the zero flag `ZF` is set, it indicates that "`count`" equals to 0; otherwise, it indicates "`count`" is larger than 0. Then we drive the control flow based on `ZF` using *conditional branch gadgets* (gadget 8-11 in Listing 1). Because the *conditional branch gadget* uses the `EAX`, `EDX`, `EBX`, and `EBP`. They may be reused in the "*loop body*". Thus, we save the registers after the *loop body*, and restore the registers before it (gadget 1, 3 in Listing 1).

Listing 1: Finite Loop Gadget

```
//restore registers 4  pop esp          jmp [esi-10h]
  1  popad              jmp [esi] //conditional branch
     xor al,62       5  pop edx       8  pushfd
     jmp [esi-10h]      cmc              jmp ebx
  2  loop body          jmp[esi-7dh] 9   pop edx
//save registers     //count--            cmc
  3  pushad          6  dec [edi]         jmp [esi-7dh]
     aam                jmp [edx]     10 or edx,ebp
     jmp [esi-70h]   7  popad             jmp [eax]
                        xor al,62     11 jmp [edx+esi]
```

3.2.5 Kernel Trapping Gadget

We leverage the *combinational gadget*, which is defined in Section 2.1, to construct kernel trapping gadget in JOP shellcode. Considering that the system call in shellcode needs to set register `EAX`, `EBX`, `ECX` and `EDX` for the system call index and arguments, we have to use the rest registers in the gadget. For this purpose, we select Gadget-8, which contains two parts: "Gadget-8a" (`libgcj.so.5.0.0`) and "Gadget-8b" (`libc-2.3.5.so`). If we need to invoke the function call, we use Gadget-8a by storing function address in the memory `[ESI+54h]` .

```
1 call [esi+54h]                        (8a)
2 jmp  [ebp-18h]

1 call large dword ptr gs:10h           (8b)
2 ret
```

3.3 Turing Completeness

The JOP gadgets, which are illustrated in this section, are as powerful as the ROP gadgets [8]. It is a hard issue to directly prove our JOP is Turing complete: being able to compute every Turing-computable function on Turing Machine [23]. However, we could find our gadget has identical functionality as ROP gadget proposed in [8]. We hence believe our JOP gadget is Turing complete.

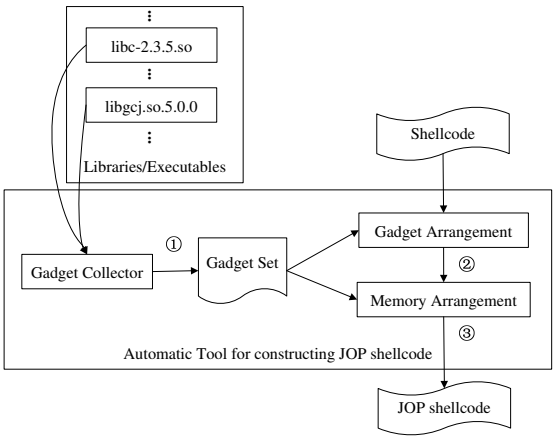

Figure 3: An Overview of Our Automatic Tool for Constructing the JOP shellcode

4. AUTOMATIC TOOL FOR JOP SHELLCODE CONSTRUCTION

4.1 Challenges

Challenge I. In a typical shellcode, attackers usually use several system calls to achieve their goal [26]. Thus, we should design a method which can achieve not only the *intra arrangement* of system call but also the *inter arrangement* between system calls. *Intra arrangement* reflects the setting of stack layout, arguments and system call index of the system call. *Inter arrangement* indicates the connection between the system calls, for example, the return value(`EAX`) of former system call will be used to set the arguments for the later one.

Challenge II. We should provide a method to solve the *data dependency* and *side effect* between gadgets. If one gadget contains the register which need to be set by other gadgets, there exists *data dependency* between them. If one gadget unintendedly sets one register and may affect the subsequent gadgets, it will cause *side effect*. Take the gadget "`mov ecx, [ebx]; mov eax, 0xffff0983; jmp [esi]`" for instance, the function of this gadget is to set the value of `ECX`. There exists a *data dependency* that `EBX` should be set by previous gadget, and also the *side effect* caused by the gadget unintendedly changing the value of `EAX`.

Challenge III. Similar to the data dependency and side effect among registers, data in memory could have dependency and side effect as well. As such, we should elaborately arrange the memory layout, and the order of the memory access of the JOP shellcode to avoid the memory conflict usage. If, for example, "`mov eax,[ebx-10h]; jmp [esi]`" and "`mov ecx,[ebx-11h];jmp edi`" are the contigeous gadgets, there exists the memory conflict usage between "`[ebx-10h]`" and "`[ebx-11h]`".

4.2 Design and Implementation

Before describing our detailed approaches to solve these challenges, we first describe the specifications of the *Shellcode* (formula-1) and the *Gadget* (formula-2).

In general, shellcode uses system calls (*"Sys_Call"*) to execute the a complicated task [26]. System call usually maintains a fixed stack layout (*Mem_U*) and register usage (*Reg_U*), *"Reg_U"* represents the registers which point to the arguments and system call index. We statistically analyzed the 130 pieces of shellcode from `milw0rm` [24], and found that the system call used by shellcode has less than four arguments. Thus the *Reg_U* contains `EAX`, `EBX`, `ECX`, and `EDX`. Note `EAX` is the system call index, `EBX,ECX,EDX` point to arguments [25]. *"Mem_U"* indicates the stack layout which contains the arguments, system call index, return value as well as the constant value. In addition, shellcode also contains other operations ("OP"), including the "data" operations (*Data_OP*) and "control" operations (Control_OP). "data" operation indicates the expression such as load/store, arithmetic/logic associated with system call. For example, in shellcode "setreuid(getuid(),getuid),execve("/bin//sh",0,0)" [27], there is a "data" movement from the return value of `setreuid` to the parameters of `getuid`. "control" operation means the control flow branch. For example, in the shellcode "Ho Detector" [28], there is a control flow branch to `exit` which depends on the return value of `read`.

A gadget is the instruction sequence which reads a set of register/memory, performs a well-defined operation on these, and then writes the result into the destination register/memory. We describe the *Gadget* in the quintuple (formula-2): (1) Gadget Functionality (*GF*); (2) Register Usage Set (*US*); (3) Register Constrain Set (*CS*); (4) Register Attach Set (*AS*); (5) Memory Usage (*MU*). *"GF"* is the functionality of the gadget, such as load gadget, store gadget and so on. To avoid possible ambiguity, one gadget has only one functionality. *"US"* is the *intended* register which is set by the gadget according to its functionality. For example, "*US*" in *load gadget* is the registers which get the value from the memory, whereas "*US*" in *inc gadget* is the register increased. *"CS"* represents the registers read by the gadget, including the registers which hold the memory address, *jump register* and so on. *"AS"* is the *unintended* registers updated in the gadget excluding the one in *"US"*. *"MU"* is the memory locations which are used by the gadget. Take *load gadget* "*mov ecx, [ebx]; mov eax, 0xffff0983; jmp [esi]*" for instance, GF={load}, US={ECX}, CS={EBX,ESI}, AS={EAX},MU={[[EBX],[ESI]]}.

4.2.1 Gadget Arrangement

Gadget arrangement contains two main tasks. One is to select gadgets to set the inter and intra arrangement of system calls. The other is to eliminate the "data dependency" and "side effect" between gadgets.

Algorithm 2 Load Gadgets Selection Algorithm

```
1: /* JOP: gadgets which have been selected; I: universal set contains the eight general-purpose registers; R_Set: the candidate
   registers needed to be set; Tmp: gadget set which contains the selected load gadgets for R_Set; Pos: the index of gadget in JOP;
   Load/Load': load gadget */
2: Load_Gadgets_Selection(JOP,R_Set,Pos){
3:   Tmp=∅;
4:   while(R_Set≠∅) {
5:     if((Load.US ∩R_Set)≠∅ ∧Load.CS⊆(I-R_Set)∧ Load.AS ⊆ (I-R_Set) ){
6:       Put Load in Tmp;
7:       if(Load'.US ⊆ Load.US ∧ Load' ∈ Tmp)
8:         remove Load' from Tmp;
9:       R_Set=R_Set-Load.US;
10:    }
11:  }
12:  Insert Tmp before JOP[Pos];
13: }
```

We select the gadgets in five steps: steps 1-3 are used to do intra arrangement of system call, step 4 is used to do inter arrangement between system calls, while step 5 is used for connecting all the selected gadgets by *control gadgets*. Specifically,

1. Check the register usage (*Reg_U*) and memory usage (*Mem_U*) of the system call, and use the "write constant to memory" gadget (Section 3.2.1) to set the constant value which has NULL bytes, including NULL arguments, system call index, string buffer and any other constant values.

2. Select the *load gadgets* (Section 3.2.1) to set the *Reg_U* of the system call in a consecutive order, including the arguments and the system call index. This can be achieved by Algorithm 2 "*Load Gadgets Selection Algorithm*". *R_Set* contains the candidate registers which need to be set and at the start, it equals to *Reg_U*. The main idea is that, if one *load gadget* can set the registers in *R_Set* and its *"AS"* and *"CS"* are both in the subset of *"(I-R_Set)"*, it has priority to be selected (line 5-6). The reason is the registers in *"(I-R_Set)"*

$$Shellcode = \{(Sys\_Call, OP) | Sys\_Call = Mem\_U \cup Reg\_U, OP = Data\_OP \cup Control\_OP\}; \quad (1)$$
$$Gadget = \{(GF, US, CS, AS, MU)\}; \quad (2)$$

are either the one which is not intended to be set or the one which has been set. Note that the candidate registers which have been set by *load gadget* (in "US") are removed from the *R_Set* and thereby in *"(I-R_Set)"* (line 9), so we can use them to set the rest candidate registers. If the later selected *load gadget* can set the registers which have been set by former gadgets, we remove the former gadgets (line 7-8). Note there may be multiple choices to select the *load gadgets*, because different *load gadgets* may set the same register, our algorithm only adopts one of them. The algorithm stops when the *"R_Set"* is empty (line 4), and we insert the selected *load gadgets* (in *Tmp*) into the gadget set *JOP* (line 12). Further, we check each selected *load gadget* and guarantee that the register in "*CS*" or "*AS*" is loaded after the register in "*US*". For example, if we select the gadget "`mov ecx,[ebx-11h];jmp [esi-39h]`" to set "`ECX`", in which the *"CS"* is `{EBX,ESI}`, and the *"US"* is `{ECX}`, we should set `EBX` after this gadget. Otherwise, the value of `EBX` will be overwritten by the value for calculating the memory `[ebx-11h]`. Algorithm 2 can be used not only for setting *Reg_U* but also in other scenarios for selecting the *load gadget* to set registers, including selecting *control gadget* to set *jump registers*.

3. Insert the "kernel trapping gadget" (Section 3.2.5) to invoke the system call.

4. Select the gadgets to achieve other operations in shellcode. For the data operation (*Data_OP*), we select the load/store gadgets and arithmetic/logic gadgets. To be more specific, we store the return value of the preceding system call to the memory, and then load the value from the memory to destination registers for the later system call (Section 3.2.1), if there are any arithmetic/logic operations, we insert the arithmetic/-logic gadgets (Section 3.2.2). For the control operation (*Control_OP*), we use the conditional branch, unconditional branch, or loop gadgets to connect the system calls (Section 3.2.3 and Section 3.2.4).

5. Select the *control gadgets* (Section 3.1) to chain the gadgets together. First, we glean the consecutive selected gadgets whose *jump registers* have not been set and divide them into the groups, each group contains no repeat gadget. For each group, we put the *jump registers* in *"R_Set"*, and adopt the Algorithm 2 to select the *control gadgets*. There are two tricks we would like to point out. One is we select *control gadget* until its *jump register* can set by itself, or its *jump register* is `ESI` which can be set by the unified *control gadgets* (Gadget-1 and Gadget-2). The other is, `ESI` is used as the *jump register* in *control gadgets* and unavailable for computation in *function gadgets*. Based on the fact that `ESI` is often used to perform string load (`LODS`) or movement (`MOVS`) in shellcode, we replace the `ESI` with other registers (e.g., `EAX`), and perform the string related operations by using the *load/store* gadgets.

Although we have selected the gadgets for the intra and inter arrangement of system call in shellcode, we encounter the second challenge: the "data dependency" and "side effect" between gadgets, and we should establish the policies to identify the data dependency and eliminate the side effect. There are two cases. The first case is, if one register belongs to the "AS" of certain gadget, after which, there is a gadget contains the register in its *"CS"*, and between these two gadgets, no gadget sets the register(in *"US"*). We insert the *load gadget* to eliminate the "side effects" between gadgets based on Algorithm 2. For example, "`mov ecx, [ebx]; mov eax, 0xffff0983; jmp [esi]`" and "`mov edx,[eax];jmp [esi-11h]`", `EAX` is in "AS" of the former gadget, and in "CS" of the later gadget, if there is no gadget between them to set `EAX`, we insert the *load gadget* immediately after the former gadget. The other case is that, if one register belongs to the "*CS*" of certain gadget, and there is no gadget to set it beforehand, then we insert the *load gadget* before the gadget to solve the "data dependency" based on Algorithm 2.

4.2.2 Memory Arrangement

After the gadget arrangement, we encounter the third challenge: there may exist the memory conflict usages between gadgets. In JOP shellcode, we fetch the address/data from stack by `MOV/POP` instruction. Except some specific usages with the same memory location, for example, to store the return value of one system call and load the value from the same memory to set the arguments of the later system call, the operations should load/store the value from different memory locations. However, the memory conflicts usages may occur at the following three cases. One case is that "`pop`" operation may conflict with the "`mov`" operation. For example, with the same `ESP`, the second "`pop reg`" conflicts with "`mov reg,[esp+4]`". Thus, we translate the "`pop reg`" into "`mov reg,[esp+offset]`", here all the stack related operands are represented in the form of "`[`esp_k`+offset]`". When the gadget changes the value of `ESP`, we will mark the old and new value of "`ESP`" as "esp_k" and "esp_{k+1}". We ensure that there are no overlaps between "`pop`" and "`mov`" operations. The second case is, we ensure the two memory locations have more than 4 bytes distance and less than the *maximum distance limit* (default 512 bytes). Otherwise, we use the *load gadget* to reset the register. For example, if two gadgets contain the memory locations which are based on the same register (e.g., "`mov eax, [esi-7dh]`; ..." and "`mov, ebx, [esi-7bh]`; ..."), and the register is set by the same gadget, we need to insert the *load gadget* to reset the register. The third case is that one register is used by two gadgets for different purpose but no gadget resets it. For example, if one register is used as operand in one gadget and used in the memory operand in another gadget ((e.g., "... ;`jmp edi`", "`mov ecx,[edi]`; ..."), or if one register is used as operand in two gadgets (e.g., "`mov ecx,edi`; ...", "... ;`jmp edi`"), we need to insert the intervening *load gadget* between the two gadgets, because the same value of the register can only satisfy with one gadget.

After eliminating the memory usage conflicts, we assign

the address and data into the memory layout of JOP shellcode. In our implementation, all the address and data are stored on the stack, therefore, the address is directly related to `ESP` (e.g., `mov ecx,[esp+4h]`) or indirectly related to `ESP`(e.g., `pop eax;mov ecx,[eax]`). We analyze all the gadgets, and translate the memory operands into the expression of `ESP`. We properly set each updated value of `ESP`, and therefore arrange the memory locations for the address and data of JOP shellcode.

5. EVALUATION

We have implemented a prototype of our tool which automatically construct JOP shellcode. Specifically, we give a user interface to define the stack layout (Mem_U) and register usage (Reg_U) of the shellcode. Then our tool will automatically select the gadgets from our gadget set and construct the JOP shellcode. We have performed our experiment on the two commonly used GNU libraries: `libc-2.3.5.so` and `libgcj.so.5.0.0`. Our testing environment is a 2.53GHz Intel Core 2 Duo CPU T9100 running Fedora Core Release 5 with linux kernel version 2.6.15.

5.1 Analysis of the Effectiveness

We choose a number of pieces of shellcode from `milw0rm` [24], and rewrite them into JOP shellcode by using our automatic tool. Table 1 shows 22 JOP shellcode our tool generated. In fact, we can automatically construct any shellcode by our tool. Based on the behaviors of these shellcode, we could classify them into three categories, *sequential*, *conditional branch* and *loop*. "*sequential*" shellcode performs their behaviors by a sequence of expressions at the unit of system call. This kind of shellcode accounts for the majority of the shellcode. "*conditional branch*" shellcode performs certain conditional branch to determine the malicious behaviors. "*loop*" shellcode often uses the loop operation to modify or decode the second stage shellcode and then jumps to that shellcode, so it is often combined with other "*sequential*" or "*conditional branch*" shellcode. As such, we only focus on the loop operation in "*loop*" shellcode. In the 130 shellcode from `milOwrm`, there are 88 *sequential* shellcode, 29 *conditional branch* shellcode, and 13 *loop* shellcode. We successfully construct all the three kinds of shellcode.

We show the *Gadget Number* of each JOP shellcodes in Table 1. It includes the *Jump Gadget* (column 5) and *Combinational Gadget* (column 6). Since the shellcode have different stack layout and register usages, we construct the JOP shellcode with different number of gadgets. Generally speaking, the *Gadget Number* is in direct proportion with the size of the shellcode, because larger shellcode has more complex stack layout and register usages. *Combinational Gadgets* (column 6) are used specifically as "kernel trapping gadgets", so the number of them equals to the number of system calls in shellcode.

We also show the size of JOP shellcode (column 7-9), including the Addr & Data (column 7), *Padding* (column 8), and *Total Size* (column 9). JOP shellcode has two parts, one part is Addr & Data, which contains the gadget address, stack address, and the constant data; it occupies most of the JOP shellcode's space, accounting for an average 75.9% of the JOP shellcodes listed in Table 1. The other part is the *Padding* which is used to fill the gap between the Addr & Data, because the gadgets often fetch the address/data from non-continuous memory locations (e.g., `[esi-77h]` and `[esi]`), and between these memory locations, we need to insert the *Padding*. The size of the *Padding* is determined by the selected gadgets and the strategy of *Memory Arrangement*. To shorten the size of the *Padding*, we can choose the smaller *maximum distance limit* between the two memory location (in Section 4.2.2) and bring more *load gadgets* to reset the register. For example, we can reset `ESI` to make `[esi-77h]` and `[esi]` be the continuous memory locations. From Table 1, we can see that the size of JOP shellcode (column 9) is larger than that of the original shellcode (column 3), and usually with a few hundreds bytes length.

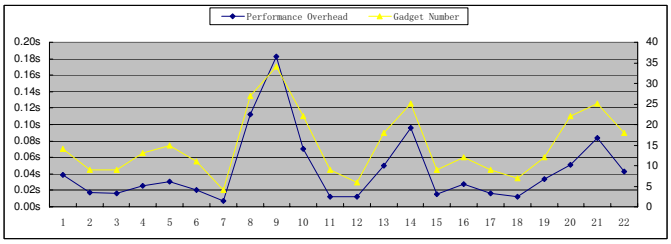

Figure 4: Performance Overhead

Unlike the traditional ROP shellcode, our method avoids executing `ret` that is not matched with `call`, thus circumventing IDSes that rely on such behavior. In our experiment, we leverage DROP [18], which can detect the traditional ROP shellcode, to evaluate our JOP shellcode. The result is JOP shellcode can bypass DROP. To the best of knowledge, there are no specific tools particularly for detecting our JOP shellcode.

5.2 Analysis of the Performance Overhead

We also analyzed the performance overhead of our tool for constructing the JOP shellcode. Note that we ignored the time costs on gadget collection. Experimental results in Figure 4 shows, within one second, our tool can generate a piece of JOP shellcode. The performance costs of JOP shellcode generation is proportional to the gadget number, about 0.003s/gadget on the average. We believe such performance overhead is acceptable. The main time costs are on gadget regulation for data dependency and side effect, and memory arrangement. In Table 1, we can see that, the longer of the shellcode, the more gadgets will be used to construct it. Because of this, the performance overhead is in direct proportion with the size of the shellcode.

There is an exception when the shellcode contains the control flow transfer. Because the control flow statement uses at least 4 gadgets (in section 3.2.3), which correspond to one conditional jump instruction in shellcode. Thus, if the shellcode contains the control flow transfer, it will suffer relatively high costs. Take the shellcode "dup2(0,0); dup2(0,1) dup2(0,2);" for instance, it contains the conditional branch and its original size is 15 bytes, 22 gadgets are used for constructing it, while the shellcode "File unlinker", which is the "*sequential*" shellcode and its original size is 18 bytes, it can be fulfilled solely with 12 gadgets. The former shellcode costs 0.051 seconds, whereas the later one costs 0.036 seconds.

5.3 Case Study

5.3.1 Launch the JOP shellcode

Similar to the ROP attack [8], we leverage software vulner-

Categories	Number	Size (bytes)	Description	JOP shellcode				
				JOP Gadget Number		JOP Shellcode Size (bytes)		
				Jmp Gadget	Comb Gadget	Addr&Data	Padding	Total Size
Sequential	1	30	chmod("//etc/shadow",666) exit(0)	12	2	264	60	324
	2	34	killall5 shellcode	8	1	188	74	262
	3	30	PUSH reboot()	8	1	186	74	260
	4	40	/sbin/iptables -F	12	1	268	32	300
	5	45	execve(rm -rf /) shellcode	14	1	317	64	381
	6	25	execve("/bin//sh", ["/bin/sh", NULL]) shellcode	10	1	216	72	288
	7	5	normal exit w/ random return value	3	1	52	120	172
	8	34	setreuid(getuid(),getuid()),execve("/bin//sh",0,0)	24	3	424	79	503
	9	86	edit /etc/sudoers for full access	30	4	604	91	695
	10	45	system-beep shellcode	20	2	360	135	495
	11	12	iopl(3); asm(cli); while(1)	8	1	116	88	204
	12	7	forkbomb	5	1	88	84	172
	13	36	write(0,"Hello core!",12)	16	2	336	120	456
	14	40	eject cd-rom (follows /dev/cdrom symlink) + exit()	22	3	412	119	531
	15	39	anti-debug trick (INT 3h trap) + execve /bin/sh	8	1	128	76	204
	16	12	set system time to 0 and exit	10	2	212	64	276
	17	11	kill all processes	8	1	160	76	236
	18	16	re-use of /bin/sh string in .rodata shellcode	6	1	132	80	212
	19	18	File unlinker	10	2	212	64	276
Conditional Branch	20	15	dup2(0,0); dup2(0,1); dup2(0,2);	21	1	268	129	397
	21	56	Ho' Detector	22	3	528	45	573
Loop	22	25	Radically Self Modifying Code	18	0	220	60	280

Table 1: Jump-Oriented Programming Shellcode

abilities to compromise the system, such as stack overflow, format string and so on. In our experiment, by leveraging the crafted stack overflow in our dedicated program, we overwrite the return address with the first gadget's address, and next to the return address is the JOP shellcode. When executing the first gadget, "esp" points to the JOP code, and it will consecutively execute the next gadget. We modify the vulnerable program in Shacham's ROP work [8]. Figure 5 shows such a vulnerable program. The buffer overflow vulnerability of this program is "`strcpy(buf,arg)`" in function "*overFlow*" (line 32). If the length of the buffer "arg" is larger than the length of "buf", it triggers the buffer overflow. We store the JOP shellcode in buffer "JOP" (line 40). To be simple, we map the libraries and stack in the fixed memory (line 41-42). The return address of *overFlow* is stored at address 0x4ffffefc. Details are shown in the source code in Figure 5.

5.3.2 Shellcode of setreuid(getuid(),getuid()),execve("/bin//sh",0,0)

In this subsection, we take shellcode "setreuid(getuid(),getuid()),execve("/bin//sh",0,0)" [27] as an instance and show how we automatically construct the JOP shellcode. There are three system calls: `getuid`,`setreuid`,`execve`. "setreuid(getuid(),getuid())" is used to set the real and effective user IDs to the values specified by the return value of "`getuid`". We first set the `EAX` as "0x31" and invoke the system call "`getuid`", then the return value of "`getuid`" is used to set the arguments `EBX` and `ECX` of "`getreuid`". To achieve the function of "execve("/bin//sh")", we put the string parameter "/bin//sh" in the shellcode. Then we set `EAX` as "0xb", `ECX` and `EDX` as the NULL pointer, `EBX` as the pointer to the memory which contains the string "/bin//sh".

Figure 6 shows the gadgets selected by our tool. For readability, we assign each gadget (could be viewed as a basic block) with a sequential number, which is different from the gadgets' index in section 3. To achieve the *Gadget Arrangement*, we follow the method in Section 4.2.1. First, we find the constants which contain zero bytes: one zero byte on stack layout (NULL terminator for the string "/bin//sh") and four word-wise data in registers (system call index "0x0000000b", "0x00000046","0x00000031", null-pointer word for the arguments of "`execve`"). Then we select the gadgets with number 2, 10, 17, 19, 21 in Figure 6 to generate the constant values into the memory. Next, according to the register usage, we select the *load gadgets* to set `EAX, EBX, ECX`, and `EDX` for each system call based on Algorithm 2. We give preference to the *load gadgets* "`pop eax; jmp [edi]`" and "`pop edx; cmc; jmp[esi-7dh]`" to set `EAX` and `EDX` respectively, because these two gadgets contain "AS" and "CS" which both belong to "{`ESI,EDI,EBP`}". For the rest two registers, we set `ECX` before `EBX` according to "`mov ecx,[ebx-11h];jmp [esi-39h]`", and set `EBX` before `EAX` according to "`pop ebx; and eax,0xff432ac4; jmp [esi]`". As a result, the order of setting registers is "`ECX, EBX, EDX, EAX`" (gadget 4 for "`getuid`", gadget 12, 13 and 14 for "`setreuid`", gadget 23, 24, 25 and 26 for "`execve`" in Figure 6). Third, we insert the *kernel trapping gadget* at the end of each system call (gadget 5, 15, 27 in Figure 6). Fourth, we select the gadgets to connect the system calls. There is a "*data dependency*" between the "`getuid`" and "`setreuid`", we use the "*store gadget*" (gadget 6 and 8 in Figure 6) to put the return value of "`getuid`" in the memory, and then load the value from memory to `EBX` and `ECX` to set the arguments of "`setreuid`" (gadget 12 and 13 in Figure 6). Fifth, we carry out Algorithm 2 and select the *control gadgets* (gadget 1 sets for gadget 2, 5-6, and 8; gadget 9 sets for gadget 10, 12-15; and gadget 16 sets for gadget 17; gadget 18 sets for gadget 19; gadget 20 sets for 21, 23-27 in Figure 6). For example, in order to set the jump registers `EDI,ESI,EBP` used by the gadgets (2, 5, 6, 8 in Figure 6), we select the gadget 1 in Figure 6, whose "US" is {`EBX,ECX,EDX,EDI,ESI,EBP,ESP`}, "AS" is {`EAX`}, and "CS" is {`ESP`}. Next, to eliminate "data dependency" and "side effect" between the gadgets, we find no *load gadgets* need to be inserted.

To achieve the memory arrangement, we follow the method in Section 4.2.2. In this example, there are seven memory conflicts (gadget 2 and 4 use `edi` for different purposes: one is for arithmetic operand, the other is for jump register, and the same reason applies to gadget 10 and 14, 21 and 26; gadget 2 and 5 both use "`ebp`" as jump register, but the former uses it directly as gadget address, and the later uses it to compute the address, the same reason also applies to gadget 10 and 15, 21 and 27; gadget 6 and 8 conflict with "`jump [esi-77h]`"), therefore we insert the *load gadgets* (gadget 3, 11, 22, 7 in Figure 6).

6. RELATED WORK

6.1 Return Oriented Programming

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <sys/mman.h>
5 #include <string.h>
6 #include <stdlib.h>
7 #include <stdio.h>
8 #define LenOfShellCode 4500  #define LenOfBuffer 4400

9 void load_Libs(void)
10{
11   int fd;
12   int fd_libgcj = 0;
13   struct stat sb;
14   struct stat st_libgcj;
15   fd = open("libc-2.3.5.so", O_RDONLY, 0);
16   fstat(fd, &sb);
17   mmap( (void*)0x03000000, sb.st_size, PROT_READ |
18            PROT_EXEC, MAP_FIXED | MAP_SHARED, fd, 0);
19   fd_libgcj = open("libgcj.so.5.0.0", O_RDONLY, 0);
20   fstat( fd_libgcj, &st_libgcj );
21   mmap( (void*)0x04000000, st_libgcj.st_size, PROT_READ
22          | PROT_EXEC, MAP_FIXED | MAP_SHARED, fd_libgcj, 0);
23}

24 void do_map_stack(void){
25    int fd;
26    fd = open("/dev/zero", O_RDONLY, 0);
27    mmap( (void*)0x4f000000, 0x01000000, PROT_READ |
28       PROT_WRITE, MAP_FIXED | MAP_PRIVATE, fd, 0);
29 }

30 void overFlow(char* arg){
31    char buf[LenOfBuffer];
32    strcpy(buf, arg);
33 }

34 void move_stack(char* arg){
35    __asm("mov $0x4fffff00, %esp\n");
36    overFlow(arg);
37    _Exit(0);
38 }

39 int main(int argc, char* argv[]){
40     char JOP[LenOfShellCode];
41     load_Libs();
42     do_map_stack();
43     move_stack(JOP);
44 }
```

Figure 5: Vulnerability Software

```
//set 0x31          //load EAX
1                   4
popad               pop eax
xor al, 62          jmp [edi]
jmp [esi-10h]
                    //getuid()
2
add [ebx],edi       5(a)
jmp ebp             call [esi+54h]
                    jmp [ebp-18h]
3
popad               5(b)
xor al, 62          call large dword
jmp [esi-10h]       ptr gs:10h
                    ret
```

getuid()

```
//store getuid() //set 0x46       //load ECX EBX,EAX //setreuid(getuid(),getuid())
6                 9               12                 15(a)
xchg eax,edi      popad           mov ecx, [ebx-11h] call [esi+54h]
cmp ah,dh         xor al, 62      jmp [esi-39h]      jmp [ebp-18h]
jmp [esi-77h]     jmp [esi-10h]

7                 10              13                 15(b)
pop esi           add [ebx],edi   pop ebx            call large dword
jmp [esi]         jmp ebp         and eax,ff432ac4   ptr gs:10h
                                  jmp [esi]          ret
8                 11
mov [ebx-2],edi   popad           14
jmp [esi-77h]     xor al, 62      pop eax
                  jmp [esi-10h]   jmp [edi]
```

setreuid(getuid(),getuid())

```
//set 0xb and two zero words   19                  //load ECX, EBX,EDX,EAX   26
                               add [ebx],edi       23                      pop eax
                               jmp ebp             mov ecx, [ebx-11h]      jmp [edi]
16                                                 jmp [esi-39h]
popad                          20                                          //execve("/bin/
xor al,62                      popad                                       /sh",0,0)
jmp [esi-10h]                  xor al, 62          24
                               jmp [esi-10h]       pop ebx                 27(a)
17                                                 and eax,ff432ac4        call [esi+54h]
add [ebx],edi                  21                  jmp [esi]               jmp [ebp-18h]
jmp ebp                        add [ebx],edi
                               jmp ebp             25
18                                                 pop edx                 27(b)
popad                          22                  cmc                     call large dword
xor al, 62                     popad               jmp [esi-7dh]           ptr gs:10h
jmp [esi-10h]                  xor al, 62                                  ret
                               jmp [esi-10h]
```

execve("/bin//sh" ,0 , 0)

Figure 6: JOP shellcode: setreuid(getuid(),getuid()),execve("/bin//sh",0,0)

Two other parallel and independent works in improving traditional ROP techniques [20,29] have been proposed. Checkoway et al. [20] proposed the techniques which construct ROP shellcode without using the return instructions. Instead of ret, they construct the ROP attack by the pop-jmp instruction sequence, which behaves like return instruction. There are two disadvantage of their work. First, they did not propose the method how to set the jump register. In particular, they frequently use the `edx` as jump register, but the gadget (e.g., `pop %edx; jmp *x`) which sets the `edx` may lead to the circle problem as mentioned in Section 3.1. Second, they do not provide the kernel trapping gadget, which is the key to construct the shellcode.

In addition, Bletsch et al. [29] proposed the technique "Jump-Oriented Programming (JOP)". There are two differences between our methods. First, they leverage the gadget ending in independent `call` instruction, which has no corresponding `ret` instruction. Existing ROP defending tools (e.g., ROPdefender [16]) can be easily modified to detect their JOP shellcode with independent `call` instruction by monitoring the imbalance in the ratio of executed call and ret. In our work, we use the *combinational gadgets* to eliminate the independent `call` or `ret`. Second, the *Dispatcher Gadgets* proposed by Bletsh et al. can connect only a few gadgets, because they do not provide the method how to set up the *jump registers* other than those in *Dispatcher Gadgets*. In this paper, we propose the *Control Gadgets* to set the *jump registers*. Moreover, neither of them ([21, 29]) provides the techniques to eliminate such as the side effect of gadget and memory usage, which are actually important techniques in JOP.

6.2 Binary Code-reuse Methods

Researchers create several interesting binary code-reuse techniques to construct the binary program without bringing new code. Caballero et al. [30] proposed a code reuse method called BCR, which extracts an assembly function from a program binary (e.g.,malware), based on which, the attacker constructs the self-contained binary program. In addition, Inspector [31] generates a so-called gadget from a binary and reuses it to achieve the malicious behavior. Lin et al. [32] proposed a new trojan construction method which reuses malicious function in a legitimate binary code and performs malicious activities. All the works mentioned above use the existing binary code, and try to find the useful code snippet to do malicious behavior. Most recently, Blazakis [33] proposes a method which uses the code dynamically generated by flash VM to construct the shellcode.

7. CONCLUSION

In this paper, we have presented a new type of shellcode which is based on Jump-Oriented Programming gadgets. Such new JOP shellcode can bypass most of the existing ROP defenses. Statically, it appears like legal program basic blocks, and dynamically it does not have the statistic behavior such as `ret` without `call`. Moreover, it can provide new polymorphism capabilities in the ending instruction compared with ROP. We have presented techniques to find out the JOP gadgets, and show these gadgets are Turing complete. Further, we have implemented an automatic tool which is capable of generating JOP shellcode. Also

we discuss that JOP can be used to construct JOP rootkit. Similar to ROP attacks, we believe JOP based attacks will be a new threat shortly.

8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive and helpful feedbacks and suggestions. 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)

9. REFERENCES

[1] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006, pp. 2–16.

[2] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX Conference on System Administration*. Berkeley, CA, USA: USENIX Association, 1999, pp. 229–238.

[3] H.-A. Kim and B. Karp, "Autograph: toward automated, distributed worm signature detection," in *Proceedings of the 13th Conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2004, pp. 271–286.

[4] "The pax project," 2004. [Online]. Available: http://pax.grsecurity.net/

[5] J. McDonald, "Defeating solaris/sparc non-executable stack protection," *Bugtraq*, 1999.

[6] Nergal, "The advanced return-into-lib(c) exploits (pax case study)," *Phrack Magazine*, 2001. [Online]. Available: http://www.phrack.com/issues.html?issue=58&id=4

[7] S. Krahmer, "X86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," *Phrack Magazine*, 2005. [Online]. Available: http://www.suse.de/krahmer/no-nx.pdf

[8] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security(CCS)*. New York, NY, USA: ACM, 2007, pp. 552–561.

[9] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 15–26.

[10] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, "Can dres provide long-lasting security?the case of return-oriented programming and the avc advantage," in *Proceedings of EVT/WOTE 2009.USENIX/ACCURATE/IAVoSS*, 2009.

[11] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings of 18th USENIX Security Symposium*, San Jose, CA, USA, 2009, pp. 383–398.

[12] 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*. New York, NY, USA: ACM, 2008, pp. 27–38.

[13] "Felix "fx" lidner.developments in cisco ios forensics," *CONFidence 2.0.*, http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf.

[14] T. Kornau, "Return oriented programming for the arm architecture," *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[15] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *SecuCode '09: Proceedings of the first ACM workshop on Secure execution of untrusted code*. New York, NY, USA: ACM, 2009, pp. 19–26.

[16] L. David, A.-R. Ssadeghi, and M. Winandy, "Ropdefender:a detection tool to defend against return-oriented programming attacks," *Technical Report TR-2010-001*, 2010.

[17] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, 2009, pp. 49–54.

[18] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code." in *ICISS*, ser. Lecture Notes in Computer Science, A. Prakash and I. Gupta, Eds., vol. 5905. Springer, 2009, pp. 163–177.

[19] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *EuroSys '10: Proceedings of the 5th European conference on Computer systems*. New York, NY, USA: ACM, 2010, pp. 195–208.

[20] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of CCS 2010*, A. Keromytis and V. Shmatikov, Eds. ACM Press, Oct. 2010, pp. 559–72.

[21] S. Checkwway and H. Shacham, "Escape from return-oriented programming: Return-oriented programming without returns(on the x86)," *Technical Report*, 2010.

[22] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Return-oriented programming without returns on arm," *Technical Report HGI-TR-2010-002, Ruhr-University Bochum*, 2010.

[23] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proc. London Math. Soc.*, pp. 230–265, 1936.

[24] milw0rm. [Online]. Available: http://www.milw0rm.com/shellcode/linux/x86

[25] "Implementing linux system calls," *Linux Journal*, 1999. [Online]. Available: http://www.linuxjournal.com/article/3326

[26] D. Mazzocchio, "Writing shellcode for linux and *bsd," 2005. [Online]. Available: http://www.shell-storm.org/papers/files/442.pdf

[27] ""linux/x86 setreuid(geteuid(),geteuid()),execve("/bin/sh",0,0)"," *milw0rm*, 2009, http://www.milw0rm.com/shellcode/8972.

[28] ""linux/x86 /ho detector"," *milw0rm*, 2008. [Online]. Available: http://www.milw0rm.com/shellcode/7154

[29] V. F. Tyler Bletsch, Xuxian Jiang, "Jump-oriented programming: A new class of code-reuse attack," *Technical Report TR-2010-8*, 2010.

[30] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, 2010.

[31] C. K. C. Kolbitsch, T. Holz and E. Kirda, "Inspector gadget: Automated extraction of proprietary gadgets from malware binaries," *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2010.

[32] Z. Lin, X. Zhang, and D. Xu, "Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction," in *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS 2010)*, Chicago, IL, USA, June 2010.

[33] D. Blazakis, "interpreter exploitation: pointer inference and jit spraying," *BHDC*, 2010, http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf.