LGadget: ROP Exploit based on Long Instruction Sequences

Jiaxin Cao[1], Tao Zheng[2+], Zhijun Huang[3], Zhitian Lin[4], Chao Yang[5]
Software Institute, Nanjing University, Nanjing, 210093, PR China
State Key Laboratory for Novel Software Technology of Nanjing University
{mg1132001[1], zt[2+], mg1032004[3], mg1232007[4], mg1232013[5]}@software.nju.edu.cn

***Abstract*—since ROP (Return Oriented Programming) has been put forward, it has broken the limit of protection strategy against malware. Finer granularity of control and complete expression make traditional defense against control flow attacks fail. In recent years, defense of ROP has made some progress. Some detection methods based on the short length of gadgets in ROP shellcode largely limit ROP. In this paper, we propose a new way of generating ROP shellcode by long instruction sequences called LGadget and design an algorithm to recognize the register dependency among instructions in LGadgets. Moreover, we build a Turing-complete instruction set based on LGadgets. By using our Turing-complete set, we construct a ROP exploit, which can break through the above-mentioned detection methods.**

***Keywords*—ROP, program security, LGadget, Turing-complete**

I. INTRODUCTION

In 2007，Hovav Shacham[1] put forward ROP technique based on Return-Into-Libc[2,3,4]. ROP breaks through the limitation that traditional Return-Into-Libc executes code orderly and removes Libc key function defense, which poses a new security threat to procedural security. ROP extracts short instruction sequences ended with a 'ret' instruction from existing object code(also called gadget) and it performs arbitrary, Turing-complete computation. As shown in Figure1, the starting address of the three short instruction sequences from glibc [23] library and two immediates are stored in stack previously.

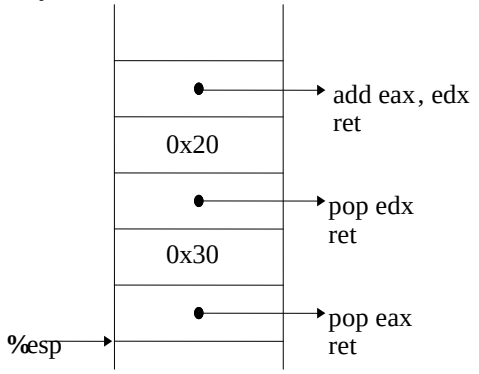


Fig.1. ROP gadgets

Program control flow executes the above three gadgets orderly according to the address information of the stack. After executing each short instruction sequence, "ret" instruction extracts the first instruction address of the next short instruction sequence from the stack and modifies the program counter %eip to implement the next short instruction sequence. Figure 1 performs 0x20 and 0x30 addition operation.

Since then, ROP exploit has been implemented on various platforms. Eric Buchanan et al[6] conduct ROP attack on RISC-based SPARC platform. Ralf Hund et al[7] achieve a semi-automatic method for constructing ROP with DLLs on Windows to conduct rootkit attack. Tim Kornau et al[8,9] conduct ROP attack on the ARM platform. Moreover, JOP[10,20,21], a ROP variant using 'jmp' instruction, expands the ROP family. The emergence of ROP and its variants make the current defenses against injection code attack fail，especially W ⊕ X(for Linux)[11,12] and DEP[13](for Windows)。

At present, many defense technologies against ROP have been developed, including reducing "ret" instruction frequency in compile phase[14], the detection based on probability statistics[15], monitoring tool based on dynamic instrumentation[16], and defense method on the base of address randomization[17]. Up to now most effective detection methods are based on the feature that the length of continuous short instruction sequences in ROP exploit is too short. Under this circumstance, we propose a ROP exploit structural scheme based on long instruction sequences (also called LGadget) and successfully construct an available Turing-complete instruction set based on glibc library. In following parts, we will introduce our algorithm and present details of our implementation. This paper makes the following contributions: (1) We put forward a method to increase the length of gadget in ROP exploit. (2) We design an algorithm to recognize the register dependency in each LGadget. (3) We build an available Turing-complete instruction set with LGadget based on glibc library.

The rest of this paper is organized as follows: Section 2 provides backgrounds of ROP technique. Section 3 indicates the relationship between LGadget and Turing-completeness, and then illustrates a G-valid algorithm to recognize the register dependency in each LGadget. Section 4 introduces our Turing-complete instruction set based on glibc library. Section 5 shows a ROP exploit constructed by our Turing-complete instruction set. Section 6 concludes this paper and introduces future work.

II. BACKGROUND

The essence of ROP is, by extracting short instruction sequences ending with "ret" instruction with a length of 2-5

from assembly codes of the object library and conducting ingenious combination of these short sequences, to construct a basic operation set to achieve the capability of Turning complete compute[5]. By hijacking normal program control flow with some frequently-used program attack method including stack overflow and format string attacks[24], ROP executes ret-ended malicious instruction sequences. Hovav Shacham[1] proves that the computation ability which is Turing-complete can be achieved as long as such short sequences are provided sufficiently. Moreover, based on glibc library, Hovav Shacham constructs a Turing-complete instruction set manually, and successfully proves its feasibility and universality. However, Ping Chen et al[15] proposes a detection scheme based on the length of short instruction sequence in ROP, which can detect most existing ROP exploit successfully.

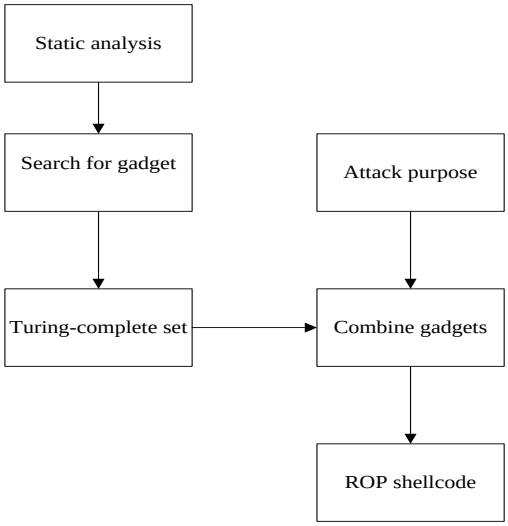


Fig.2. current ROP construct process

The current ROP exploit is constructed as Figure 2 illustrates. First we disassemble the object library and search for ret-ended instruction sequences which will be stored with a certain data structure. Before we start to build Turing-complete set, we should perform static analysis of each Turing function to combine instruction sequences to achieve the Turing function according to functional equivalence previously. Then we search for corresponding gadgets to build a Turing-complete instruction set and combine gadgets according to our purpose. For example, we combine these gadgets to perform a shell script call. In order to execute "/bin/sh", we should find the corresponding gadgets and eliminate unqualified gadgets. After that, we set variables of instruction process to implement an available ROP exploit[1]. Moreover, we need to combine the first instruction address of these gadgets reasonably to set the data distribution of the stack or heap. Finally, ROP guides the execution of control flow according to address and data distribution of the stack or heap to implement the computation of specific behavior.

III. Algorithm and Solution

A. Turing-completeness, Turing equivalence and ROP

A computational system that can compute every Turing-computable function is called Turing-complete (or Turing powerful). Alternatively, such a system is one that can simulate a universal Turing machine. A Turing-complete system is called Turing-equivalent if every function it can compute is also Turing computable; i.e., it computes precisely the same class of functions as Turing machines do. Alternatively, a Turing-equivalent system can simulate, and be simulated by, a universal Turing machine[5]. Hovav Shacham has built a gadget set by combining code segment found in libc on the Intel x86 platforms which satisfies Turing-complete expression capabilities. Those gadgets can be used to compute following basic operations, including Load/Store, arithmetic and logic operations, control flow transfer (unconditional jump, conditional jump, system calls and so on). A gadget set containing all these operations has been proved Turing-complete [1]. If the gadget set we build in this paper can achieve all the above basic operations, in other words, is Turing-equivalent with the Turing-complete set constructed by Hovav Shacham. We can deduce that the gadget set is Turing-complete.

B. LGadget and Turing-completeness

In most existing ROP shellcodes, several gadgets with length of 2-5 are combined to achieve a single Turing function. However, ROP exploit constructed by these gadgets can be easily detected due to an excessive number of gadgets and the length is too short. Ping Chen et al[15] propose a detection method, which judge that a piece of code containing more than three continuous gadgets with the length no larger than 5 is ROP exploit. The experimental results show that this detection method has a high success rate. Therefore, ROP payload constructed by above Turing-complete instruction set can hardly break through this detection method.

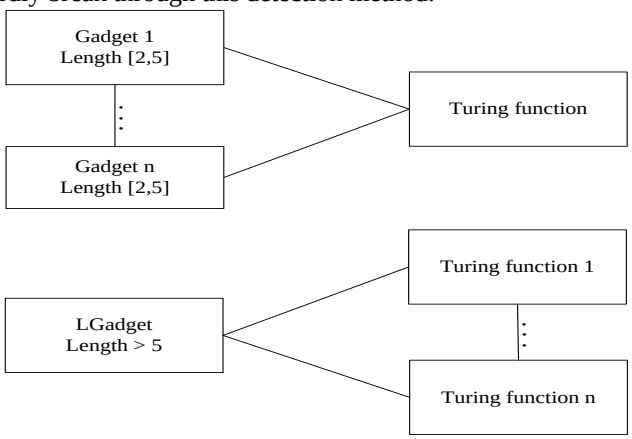


Fig.3. Gadget, LGadget and Turing function

To solve the above-mentioned problem, we propose a ROP structure scheme based on long instruction sequences. As is shown in Figure 3, we search for instruction sequences with the length more than 5, which is called LGadget, from glibc library to build a LGadget set which is Turing-equivalent with the Turing-complete set constructed by Hovav Shacham[1]. We

ensure that a single LGadget can achieve more than one operation of Turing-complete set. So ROP exploit built by LGadgets will barely include instruction sequence with length less than 5. Thus we can deduce that ROP payload based on our Turing-complete set can successfully break through the above detection method.

C. Register Dependency between instructions in LGadget

We need to screen candidate LGadgets founded from glibc to get enough available LGadgets to build Turing-complete instruction set. So we should make rules to recognize the register dependency between instructions in each candidate LGadget.

Referring to the standard 0x86 assemble instruction sets[25], all instructions in this essay are represented in this format: *opcode source operand, destination operand*, such as "add eax, ecx"**.** This instruction perform an add operation between %eax and %ecx, using %ecx as source operand and %eax as destination operand. To illustrate our algorithm, we give several definitions as follows:

Definition 1 O-Complete: If an instruction opcode belongs to the opcodes of Turing-complete instruction set constructed by Hovav Shacham[1], we call it *O-Complete.*

Definition 2 I-Accept: If the opcode of an instruction is *O-Complete* and the operands are not stack pointer, base pointer, segment and immediate, this instruction is *I-Accept*. If instruction is not *I-Accept***,** we define it as *I-Refuse*.

Definition 3:

Read(*Inst*)**:** A set recording registers loaded by the source operand or destination operand of *Inst*.

Write(*Inst*)**:** An set recording registers appear in destination operand with register direct addressing of *Inst*.

Definition 4 LGadget: An instruction sequence like <$Inst_1$, $Inst_2$, …, $Inst_n$> (n > 5, $Inst_n$ is *ret*).

Definition 5:

Suppose we have a *LGadget*<$Inst_1$, $Inst_2$, …, $Inst_n$> (n > 5, $Inst_n$ is *ret*).Then:

Pre_Write(*Inst*$_i$)**:** *Pre_Write*(*Inst*$_i$) = $\cup_{j=1}^{j<i} Write$ (*Inst*$_j$)**;**

Post_Write(*Inst*$_i$)**:** *Post_Write*(*Inst*$_i$) = *Pre_Write*(*Inst*$_n$) - *Pre_Write*(*Inst*$_{i+1}$)**;**

Pre_Write(*Inst*$_i$) and *Post_Write(Inst*$_i$) are figured out in Algorithm 1:

```
Input: LGadget G, int i, int maxlength
Output: Pre_Write(Inst_i)
place every instruction of LGadget in an array Inst[];
Procedure Prewrite(int i, Inst[]):
    for pos from 0 to i -1 do:
        Pre_Write(Inst[i]) = ∪_{pos=0}^{pos<i} Write(Inst[pos]);
        return Pre_Write(Inst[i]);
Procedure Postwrite(int i, Inst[]):
    Post_Write(Inst[i]) = call Prewrite(maxlength,
     Inst[]) – call Prewrite(i + 1, Inst[]);
    return Post_Write(Inst[i]);
```

Algorithm 1: The G-Write Algorithm

Definition 6:

*I-Valid***:** instruction $Inst_i$ in *L-Gadget* is *I-Valid* if it satisfies the following conditions:

a. $Inst_i$ is *I-Accept*
b. *Write*(*Inst*$_i$) ∪*Read*(*Inst*$_i$) ∩*Pre_Write*(*Inst*$_i$) = ∅;
c. *Write*(*Inst*$_i$) ∩*Post_Write*(*Inst*$_i$) = ∅;

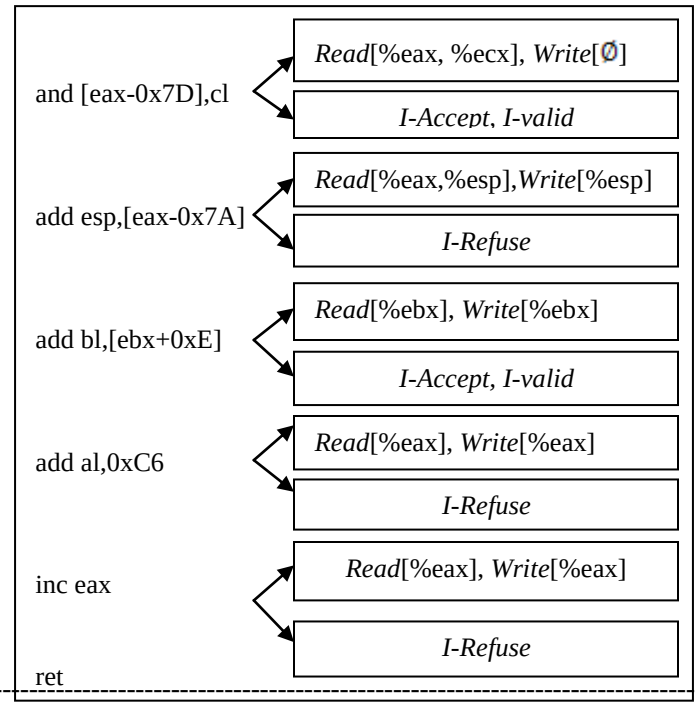
G-Valid: A *LGadget* is *G-Valid* if there exists at least one *I-Valid* instruction in it. Then we can show the whole process in Algorithm2.

:

```
place every instruction of LGadget in an array Inst[];
 int countvalid = 0;
for pos from 0 to Inst_len do:
        record the Read(Inst[pos]) and Write(Inst[pos]).
        if Validcheck(pos, Inst[]) = true, then:
                countvalid ++;
                mark Inst[pos] as I-valid;
if countvalid > 0 return true;
Procedure Validcheck(index pos, instructions Inst[]):
    create set PreWrite= call PreWrite(pos,Inst[]);
               PostWrite= call PostWrite(pos,Inst[]);
    if Inst[pos] is I-Accept, then:
        if ((Read(Inst[pos]) ∪ Write(Inst[pos]) ∩
    PreWrite = ∅) &&(Write(Inst_i) ∩ PostWrite= ∅)
            return true:
```

Algorithm 2: The G-Valid Algorithm

Take Figure 4 as an example, input is a candidate LGadget with length 6. Figure 4 records all information of the LGadget. Algorithm 2 traverses all $Inst_i$ to recognize the register dependency in this LGadget. Algorithm 2 judges that there exist two *I-Valid* instructions in this LGadget, thus this LGadget is *G-valid* and can successfully perform an ADD operation and an AND operation. LGadget in Figure 4 can achieve two operations of Turing-complete set. As a matter of fact, if we can find enough LGadgets like that to perform all operations of Turing-complete set, we will get an available instruction set for constructing ROP exploit.



D. Turing-complete instruction set construction process

The whole process is shown in Figure 5. Firstly, we disassemble the object library with libdisasm[22], and construct a trie tree based on ret-ended instruction sequences. Here we use Galieo algorithm[1] to generate the trie tree. Secondly, we search for corresponding candidate LGadgets from the trie tree based on Turing-complete set provided by Hovav Shacham[1]. Finally, we eliminate LGadgets with register dependency based on our following algorithms to generate the final Turing-complete set.

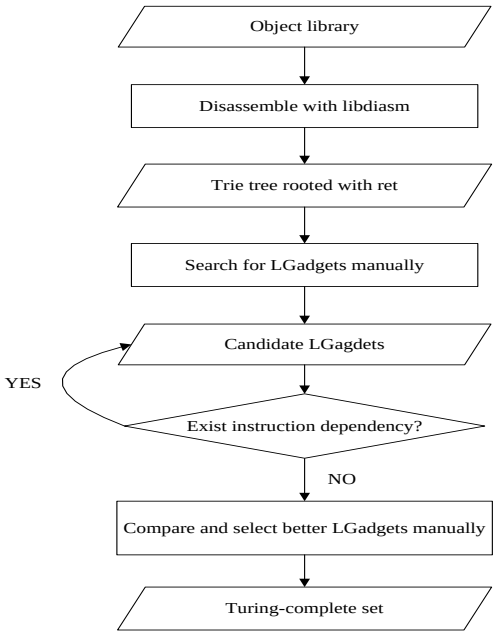


Fig.5. Flow chart of building Turing-complete set

IV. TURING-complete INSTUCTION SET

We build our Turing-complete instruction set on the gnu C Library distributed with Ubuntu Core Release 12.10: glibc 2.15. First we disassemble the target code with open source library libdisasm[22] and construct a trie tree whose root represents the "*ret"* instruction using the Galieo[1] algorithm. Secondly, based on this trie tree, we carry out our experiments observing and building long instructions sequences which can perform well-defined operations, such as "load", "xor", or "jmp". We call these sequences LGadgets. Referring to the Turing-complete gadget set in Hovav Shacham[1], our LGadgets should be able to perform a catalogue of actions: Load/Store, Arithmetic and Logic, Control Flow(unconditional jump, conditional jumps and system call).

A. Load/Store

In this paper we consider three cases: loading a constant into a register; loading the contents of a memory location into a register; and writing the contents of a register into a memory.

Loading a constant. When constructing ROP exploit stack, the variables should be stored in the stack in advance. After that, the variables can be loaded into %eax using instructions in the form of "pop eax". As shown in Figure 6, the decimal numbers in front of each instruction are the offsets of the instructions in libc.so, and the last three "pop" instructions load the variables into %ebx, %esi, and %edi respectively.

Loading from Memory. By using instructions like "mov eax, [ecx + 0x20]", the content of the specified memory address can be loaded into the register.

Storing to Memory. As shown in Figure 6, the second instruction "mov [edi], eax" successfully writes the content of %eax to the memory "[edi]". To sum up, the sequences in Figure 6 achieves three functions, which are shift logical right, writing the content of the register to the memory, and loading the variables to the register respectively.

197222: shr eax, cl	⟶ *I-Accept*	⟶ *I-valid*
197224: mov [edi], eax	⟶ *I-Accept*	⟶ *I-valid*
197226: pop eax	⟶ *I-Refuse*	
197227: pop ebx	⟶ *I-Accept*	⟶ *I-valid*
715236: pop esi	⟶ *I-Accept*	⟶ *I-valid*
715237: pop edi	⟶ *I-Accept*	⟶ *I-valid*
197230: ret		

Fig.6. Shr+Storing+Loading operations

B. Arithmetic and Logic

Add. The first instruction "add [ebx], dh" in Figure 7 serves as an add example. It adds the contents of %dh and the data at "[ebx]" to memory "[ebx]". We load two operands in advance using the load/store procedure above.

Other Operations. All other arithmetic operations such as Sub, Mul, Div, And, Or, Not, Xor, Shifts and Rotates are included in our Turing-complete set. For instance, the first instruction in Figure 6 performs a shr action. As we have already found those instruction sequences satisfying the specific needs in the glibc library, no more details are provided here.

1695751:add [ebx], dh	⟶	*I-valid*
1695753:add eax, 0x01AF0000	⟶	*I-Refuse*
1695758: add [eax], al	⟶	*I-Refuse*
1695760: add [eax], al	⟶	*I-Refuse*
1695762: add [eax], al	⟶	*I-Refuse*
1695764: ret		

Fig.7. add operation

C. Control Flow

The program control flow includes unconditional and conditional jumps.

Unconditional jump. The "ret" instruction can be used to replace the "jmp" instruction by simply putting the address of the instruction needed into the stack, by doing which, an unconditional jump can be achieved.

Conditional jump. Conditional jump instruction like jcc changes the instruction pointer to achieve a jump, so they are not useful for ROP. In order to accomplish it, a functional-equivalent gadget set should be built. Besides that, several problems including the selection

of flag register, the format of the instruction that transfers the control flow according to the flag register, and the representation of the %esp offset in the stack when the program is run needs to be considered. In this paper, CF is used as the judgment condition before a jump operation, and the starting address of the two branch instruction sequences are placed into the stack. By doing this, a conditional jump can be achieved by adding an offset to %esp. As shown in Figure 8, the whole process is as followed. Firstly, CF is set by sub operation, and assigned to all 1 or 0 in binary by neg operation. Secondly, esp_delta is placed into the stack, and an 'and' operation between its value and the present CF is performed to get a final esp_delta. Finally, this value is added to %esp to achieve a conditional jump. Because the length of each gadget cannot always be larger than 5 in this process, we place some useless gadgets between useful ones to ensure that there does not exist contiguous three gadgets with length no larger than 5. This method is also applied to ignore some Read-only memory when we build our gadgets sequences.

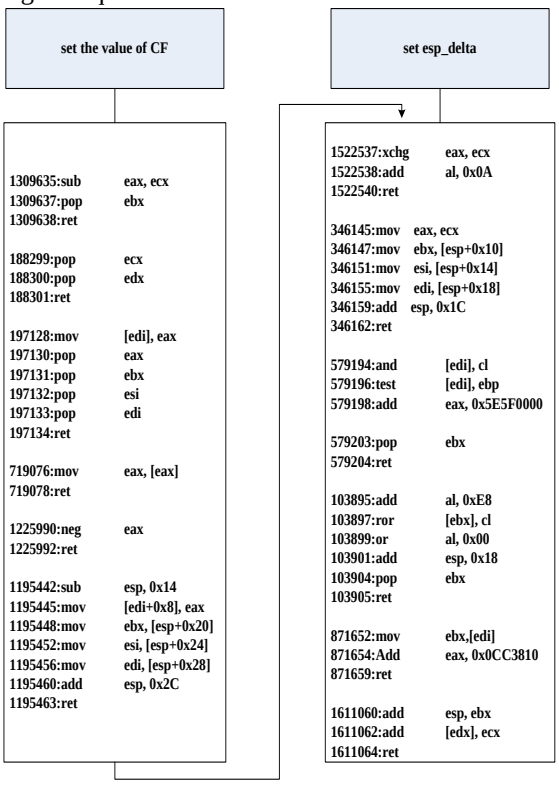


Fig 8. Conditional jump

D. System Calls

In linux, many system calls behave like this: The syscall number is set in %eax and then instruction like "lcall %gs:0x10" or "int 0x80" is executed. Since most of the syscall numbers are available in libc.so, we can finish the system calls using two methods:

Place syscall number into stack and load it to %eax using instruction like pop %eax.

Construct a system call set which consists of all discovered system call sequences in libc.so.

V. ROP SHELLCODE ANALYSIS

We will take a c language source code below for example.

int c = 0;

if(a > b) c = a – b;

else c = a + b;

Variables a and b are user input, which should be placed into the stack previously. Firstly, we need to search for appropriate LGadgets from our Turing-complete set according to the function. Secondly, we concatenate LGadgets reasonably and stuff some register initialization instruction sequences among LGadgets. Finally, we build a ROP exploit stack as shown in Figure 9 with the above-mentioned LGadgets.

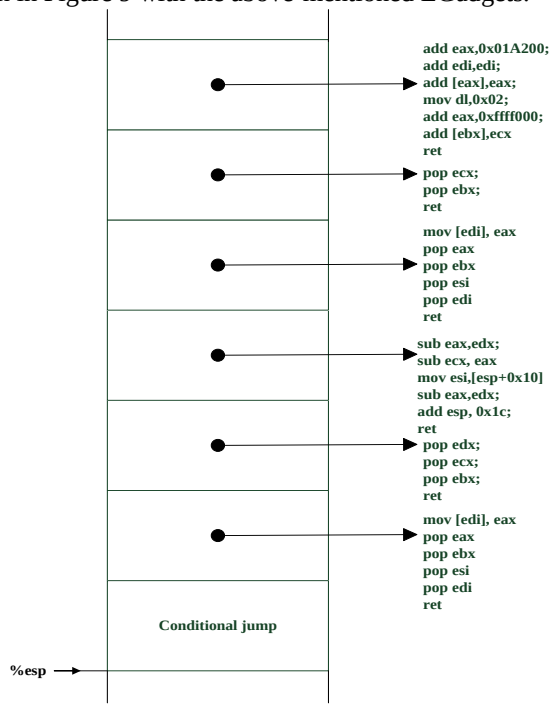


Fig.9. ROP exploit stack

As illustrated Figure 9, we use conditional jump sequences shown in Figure 8. As it can be seen from the ROP shellcode, there does not exist three contiguous instruction sequences with length no larger than five. Thus we can judge that our ROP shellcode can successfully break through detection method based on the length of gadgets.

We map libc.so into memory and specify the starting address of glibc library as 0x5656c000. After that we can get the absolute address of each instruction of our shellcode. Finally, we place the address of first instruction in each LGadgets into the stack and implement ROP exploit by string format attack. The experiments show that our ROP shellcode is effective. In addition, we have accomplished 100 ROP shellcodes, all of which are proved available, to realize different functions with our Turing-complete instruction set. In conclusion, our Turing-complete set based on LGadgets is feasible.

VI. Conclusion and Future Work

We present a new way of building Turing-complete instruction set. By selecting available long instruction sequences(also called LGadget) from the glibc library, we built a Turing-complete set which can successfully break through some detection methods based on the length of gadgets. However, there are some limitations. (1) Considerd that there exists register dependency among LGadgets, we have to insert some register initialization sequences into ROP shellcode. With the code size becoming bigger, the number of register initialization sequences becomes astronomical. (2) Some operations like conditional jump contain instructions no larger than five. (3) The whole process of building Turing-complete set is semi-automated. Unlike traditional way of building Turing-complete set, we have to spend much time recognizing register dependency with our algorithm and selecting better LGadgets manually. So, the process of building an available Turing-complete set based on LGadgets is time-consuming. There are several directions for future work.

The first direction is to optimize our Turing-complete set. We can hardly find all LGadgets with length larger than 5 from a single library or executable. But by mapping several libraries or executables into memory, we can get enough LGadgets to build a Turing-complete set with longer LGadgets and less register initiate sequences when constructing ROP shellcode. So we will search for LGadgets from more libraries or executables like Acroread.

The second direction is to attempt a more efficient process to build our Turing-complete set. Given several libraries or executables, we will try to design a method with higher degree of automation to generate an available Turing-complete set.

The third direction is to design an automation tool to conduct a ROP exploit with existing Turing-complete set. We will analyze the instructions in LGadgets and get detailed information of each instruction, which can be used as basis for combining LGadgets to construct ROP shellcodes. With an automated tool, we can construct ROP shellcodes more efficiently and get experiment samples more easily.

VII. Acknowledgment

We thank all members of our lab for providing significant advices for our work. This work was supported by the Chinese National Natural Science Foundation (NSFC 61073027, NSFC 61105069) and State Key Laboratory for Novel Software Technology of Nanjing University.

References

[1] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86): CCS'07 Proceedings of the 14th ACM conference on Computer and communications security, 2007[C]. New York, NY, USA: ACM, 2007:552-561.

[2] Weiliang Du. Return-to-libc Attack Lab[J], 2007. http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Return_to_libc/Return_to_libc.pdf.

[3] Nergal. Advanced return-into-lib(c) exploits (PaX case study)[J].http://www.phrack.org/issues.html?issue=58&id=4&mode=txt.

[4] Sebastian Krahmer, x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique[R], 2005. http://packetstorm.igor.onlinedirect.bg/papers/bypass/no-nx.pdf.

[5] Martin D.Davis, Ron Sigal, Elaine J.Weyuker. Computability, Complexity, and Languages (Fundamentals of Theoretical Computer Science, Second Edition),145-169, Posts & Telecom Press.

[6] Erik Buchanan, Ryan Roemer, Hovav Shacham. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC: CCS'08 Proceedings of the 15th ACM conference on Computer and communications security, 2008[C]. New York, NY, USA: ACM , 2008: 27-38.

[7] Ralf Hund, Thorsten Holz, Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms: SSYM'09 Proceedings of the 18th conference on USENIX security symposium, 2009[C]. CA, USA: USENIX Association Berkeley, 2009: 383-398.

[8] Tim Kornau. Return Oriented Programming for the ARM Architecture[C],http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf, Master thesis, Ruhr-University Bochum, Germany. 2009.

[9] Lucas Davi, Alexandra mitrienkoy, Ahmad-Reza Sadeghi etc. Return-Oriented Programming without Returns on ARM: In Technical Report HGI-TR-2010-002, 2010[C]. Ruhr University Bochum, Germany ,2010.

[10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, etc. Return-Oriented Programming without Returns. CCS '10 Proceedings of the 17th ACM conference on Computer and communications security, 2010[C]. New York, NY, USA: ACM, 2010: 559-572.

[11] http://en.wikipedia.org/wiki/Data_Execution_Prevention.

[12] Documentation for the PaX Project http://pax.grsecurity.net/docs/.

[13] PaX non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt.

[14] Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace etc. Defeating Return-Oriented Rootkits With ―Return-less‖ Kernels: EuroSys '10 Proceedings of the 5th European conference on Computer systems,2010[C]. New York, NY, USA ACM,2010: 195-208.

[15] Ping Chen, Hai Xiao, Xiaobin Shen etc. DROP: Detecting Return-Oriented Programming Malicious Code. In A. Prakash and I. Gupta, editors, Fifth International Conference on Information Systems Security (ICISS 2010), volume 5905 of Lecture Notes in Computer Science, Springer, 2009:163—177.

[16] Lucas Daviy, Ahmad-Reza Sadeghiy, Marcel Winandyz. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks: ASIACCS '11 Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security,2011[C]. New York, NY, USA ACM,2011:40-51.

[17] Vasilis Pappas,Michalis Polychronakis,Angelos D.Keromytis.Smashing the Gadgets:Hidering Return-Oriented Programming Using In-Place Code Randomization. Columbia University,2012.

[18] Thomas Dullien,Tim Kornau,Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. University of Luxembourg.

[19] Edward J. Schwartz, Thanassis Avgerinos and David Brumley.Q: Exploit Hardening Made Easy. Carnegie Mellon University, Pittsburgh, PA.

[20] Ping Chen, Xiao Xing, Bing Mao, Li Xie, Xiaobin Shen, Xinchun Yin. Automatic Construction of Jump-Oriented Programming Shellcode (on the x86). ASIACCS'11,March 22-24,2011, HongKong, China.

[21] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, Zhenkai Liang.Jump-Oriented Programming: A New Class of Code-ReuseAttack. ASSIACCS'11, March22-24,2011,HongKong,China.

[22] disasm. http://bastard.sourceforge.net/libdisasm.html.

[23] Glibc. http://lfs.linuxsir.org/doc/lfscvs/appendixa/glibc.html

[24] David Litchfield. Windows 2000 Format String Vulnerabilities. http://www.atstake.com/.

[25] Muhammad Ali Mazidi, Janice Gillispie Mazidi, Danny Causey. The x86 PC Assembly Language, Design, and Interfacing(Fifth Edition), 45-71, Publishing House of Electronics Industry.