ISSN 0361-7688, Programming and Computer Software, 2019, Vol. 45, No. 8, pp. 35–46. © Pleiades Publishing, Ltd., 2019.
Russian Text © The Author(s), 2019, published in Programmirovanie, 2019, Vol. 45, No. 00000.

A Method for Analyzing Code-Reuse Attacks

A. V. Vishnyakov[a,*], A. R. Nurmukhametov[a,], Sh. F. Kurmangaleev[a,***], and S. S. Gaisaryan[a,b,****]**

[a] *Ivannikov Institute for System Programming, Russian Academy of Sciences, Moscow, 109004 Russia*

[b] *Moscow State University, Moscow, 119991 Russia*

**e-mail: vishnya@ispras.ru*

***e-mail: nurmukhametov@ispras.ru*

****e-mail: kursh@ispras.ru*

*****e-mail: ssg@ispras.ru*

Received February 13, 2019; revised March 29, 2019; accepted March 29, 2019

Abstract—Nowadays, ensuring software security is of paramount importance. Software failures can have significant consequences, and malicious vulnerability exploitation can inflict immense losses. Large corporations pay particular attention to the investigation of computer security incidents. Code-reuse attacks based on return-oriented programming (ROP) are gaining popularity each year and can bypass even modern operating system protection mechanisms. Unlike ordinary shellcode, where instructions are placed sequentially in memory, a ROP chain consists of multiple small instruction blocks (called gadgets) and uses the stack to chain them together. This makes the analysis of ROP exploits more difficult. The main goal of this work is to simplify reverse engineering of ROP exploits. A method for analyzing code-reuse attacks that allows one to split the chain into gadgets, restore the semantics of each particular gadget, and restore the prototypes and parameter values of the system calls and functions invoked during the execution of the ROP chain is proposed. The semantics of each gadget is determined by its parameterized type. Each gadget type is defined by a postcondition (Boolean predicate) that must always be true after the gadget execution. The proposed method was implemented as a software tool and tested on real-world ROP exploits found on the Internet.

DOI: 10.1134/S0361768819080061

1. INTRODUCTION

Nowadays, ensuring software security is of paramount importance. To make software secure, static analysis [1], dynamic analysis [2], and their combinations [3] are used in the stage of software development. Software is widely used in everyday life in such devices as computers, smart phones, cars, ATMs, city infrastructure objects, medical equipment, and Internet-of-things technologies. Software failures can have significant consequences, such as financial losses, degradation communication means, delays in the operation of emergency services, and even harm health. Moreover, malicious vulnerability exploitation can inflict huge losses. According to the National Institute of Standards and Technology, thousands descriptions of new vulnerabilities are yearly published in the Common Vulnerabilities and Exposures (CVE) (see Fig. 1) [4, 5]. Large corporations pay particular attention to the investigation of computer security incidents.

To exploit vulnerabilities and bypass modern operating systems protection mechanisms, the return-oriented programming (ROP) is often used. ROP is a code-reuse technique that allows one to bypass the protection mechanism called data execution prevention (DEP) that does not allow a memory region be simultaneously available for write and execution. ROP is also useful to bypass modern implementations of the address space layout randomization (ASLR) that leave a part of the address space non-randomized. For instance, in Linux the program code base address often remains constant, and some Windows DLLs are loaded at fixed addresses. An intruder uses pieces of code from the non-randomized program address space; these pieces are called gadgets. Each gadget performs certain computations (e.g., adds the values of two registers) and transfers control to the next gadget. Gadgets are connected into a chain of sequentially executed pieces of code. Thus, using a chain of gadgets, one can perform malicious actions.

The application of fine-grained ASLR [6] improves the stability of the system under ROP attacks; however, the vast majority of operating systems currently use granulation up to a module.

A gadget is an instruction sequence that is ended by the control transfer instruction (`ret`). Unlike an ordinary program, ROP instructions are not placed

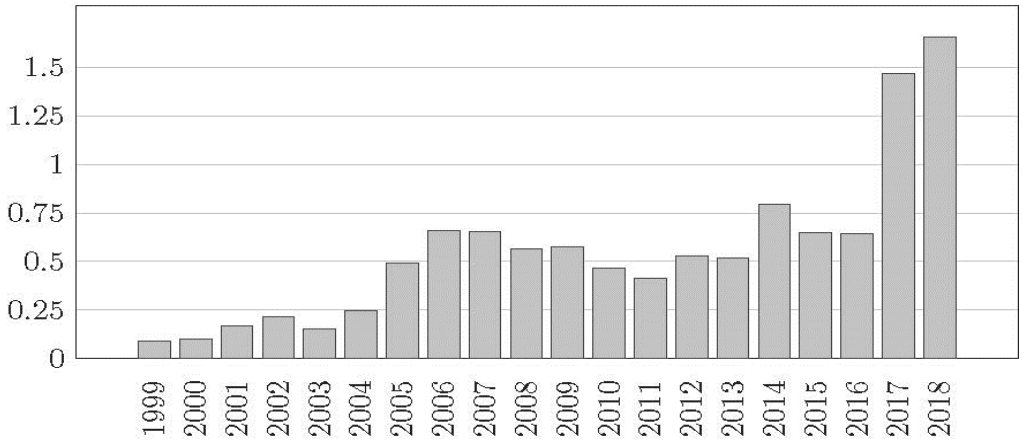

Fig. 1. Tens of thousands of vulnerabilities (CVE) by year.

sequentially in the memory, but are decomposed into small pieces of code called gadgets. Gadgets are connected by instructions that get the address of the next gadget from the stack. This technique complicates the analysis of ROP chains.

An exploit is a program, input data, or an instruction sequence that use a software vulnerability to make the program behave in an unspecified fashion. The aim of this paper is to simplify the reverse engineering of ROP exploits.

We propose a method for analyzing code-reuse attacks that helps restore the semantics of a ROP chain, i.e., decompose the chain into gadgets, determine the semantics of individual gadgets, and recover the functions and system calls, as well as their arguments, invoked during the chain execution.

The paper is organized as follows. In the second section, we review the attack techniques and protection mechanisms that were precursors of ROP (Subsection 2.5). In Section 3, we discuss the known methods of analyzing ROP attacks. In Section 4, we describe the proposed method for analyzing code-reuse attacks. In the fifth section, we discuss implementation details of this method. In Section 6, we present practical application results.

2. REVIEW OF ATTACKS AND PROTECTION MECHANISMS

In this section, we review the stack buffer overflow attacks. We describe protection mechanisms used in operating systems, such as data execution prevention (DEP) and address space layout randomization (ASLR). In subsection 2.5, we define return-oriented programming, which is a technique of exploiting stack buffer overflow; it allows one to bypass DEP and modern implementations of ASLR.

2.1. Stack Buffer Overflow

The stack buffer overflow vulnerability occurs when the size of data written to a buffer on the stack exceeds the size of this buffer [7]. For example, in the C program below, the vulnerable function `vul` does not check the length of the string `str`, which is written to a buffer `buf` of fixed size on the stack. If the length of the first command line argument `argv[1]` is greater than or equal to the size of `buf`, then the stack buffer overflow occurs.

```
void vul(char *str) {
  char buf[512];
  strcpy(buf, str);
}
int main(int argc, char *argv[]) {
  vul(argv [1]);
  return 0;
}
```

Figure 2a shows the stack frame of the function `vul` before overflow. In the x86 architecture, the stack grows from greater addresses to lower ones (from the top to the bottom in the figure). The arguments of a function are pushed on the stack one-by-one from the right to the left. As the function is called, the return address is pushed on the stack, after which the function can store the previous value of the `ebp` register and allocate memory on the stack for local variables (in the case under examination, for the buffer `buf`). The data are written to the buffer in the order of growing addresses (from bottom to top in the figure). As a result of the buffer overflow, memory locations above, including the return address, are rewritten; after this, the program almost always crashes.

The exploitation of the stack buffer overflow vulnerability allows the intruder to execute an arbitrary code. Consider the situation when an intruder controls the values of the first command line argument `argv[1]` and, therefore, the values written to `buf`. In this case, the intruder can rewrite the return address

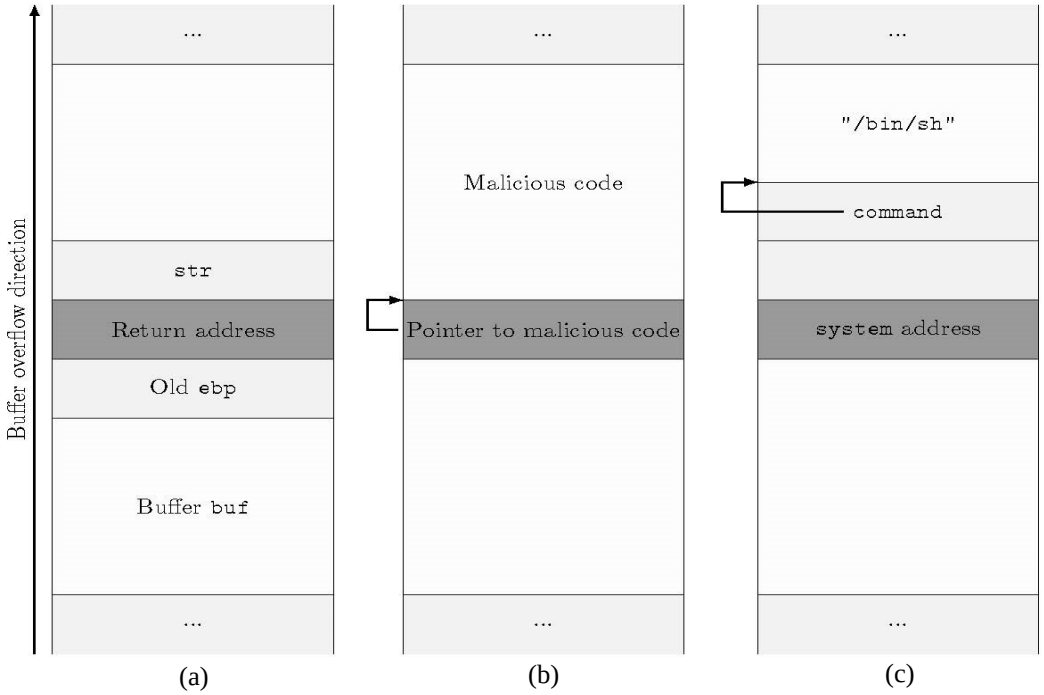

Fig. 2. Stack frame of the function `vul` and different buffer overflow exploitation techniques. (a) Stack frame of the function `vul` before overflow. (b) Location of the malicious code on the stack. (c) Return-to-libc attack.

by a pointer to a malicious code (Fig. 2b). Thus, after returning from the function `vul`, the control will be transferred to the malicious code formed by the intruder. Typically, this code opens the command shell of the operating system. That is why it is called a shellcode. To avoid harmful consequences of stack buffer overflow, various protection mechanisms were invented.

2.2. Data Execution Prevention (DEP)

The data execution prevention (DEP) is a protection mechanism in the operating system that prevents applications from executing code in the memory regions containing data. An attempt to execute code in such a region throws an exception and causes the program crash. In particular, the code on the stack and in the heap cannot be executed, which prevents the execution of malicious code in them. This technique is successfully used in Windows, Linux, and other operating systems.

2.3. Return-to-libc Attack

The return-to-libc attack can be used to bypass DEP. The idea underlying this kind of attack is to replace the return address with the address of a library function, e.g., the function `system` from the library `libc`.

Figure 2c shows the state of the stack after overflow. The return address is rewritten by the address of the function `system(const char *command)`. Above it, there is an arbitrary return address from the function system and its only argument, which is the pointer to the null-terminated string `"/bin/sh"` that resides above the pointer on the stack. Therefore, after the return from the function `vul`, the library function `system("/bin/sh")` will be called, which in turn will open the command shell of the operating system.

2.4. Address Space Layout Randomization (ASLR)

The address space layout randomization (ASLR) is an operating system protection technique that makes it possible to place the key elements of the process (program image, stack, heap, and dynamic libraries) at different addresses when the executable file is loaded. This technique complicates the execution of return-to-libc attacks because the addresses of library functions are not known before the program is loaded and they vary for different runs.

Note that the randomization of addresses of executable sections of a program or library requires that they are compiled into a position-independent code, which is not always done. For instance, in Linux the program code is often loaded to a constant address, and some Windows dynamic libraries are loaded to fixed addresses. Therefore, under modern implementations of ASLR, a part of the program address space remains non-randomized.

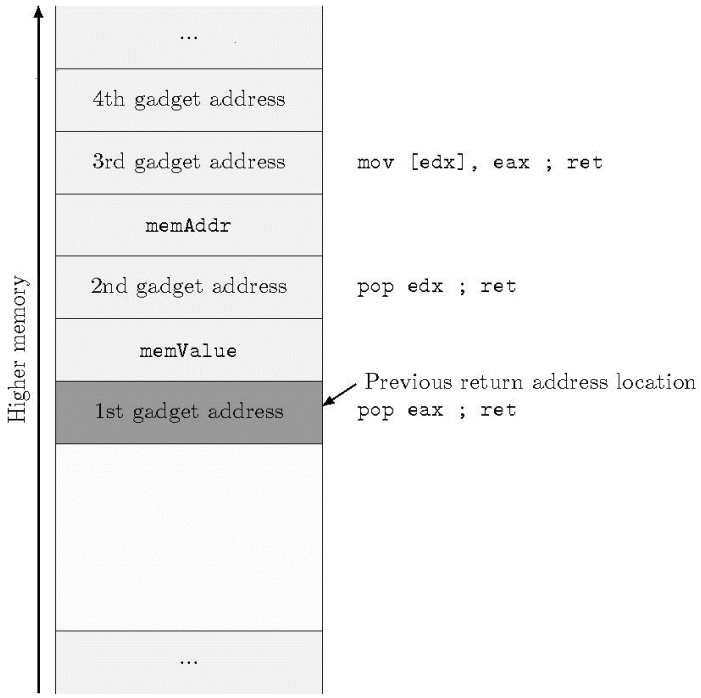

Fig. 3. The state of the stack after the ROP chain storing `memValue` to `memAddr` is written on the stack.

2.5. Return-Oriented Programming (ROP)

The return-oriented programming (ROP) [8] is a stack buffer overflow exploitation technique; in essence, this is a generalization of the return-to-libc attack. This technique can also be used to bypass DEP; however, it is more dangerous because it can be used to bypass modern implementations of ASLR when a part of the address space remains non-randomized (Subsection 2.4).

ROP is based on using sequences of instructions in non-randomized executable memory regions that end in the control transfer instruction (`ret`). Such sequences of instructions are called gadgets. Note that the x86 architecture does not require the instruction addresses to be aligned; i.e., it allows the execution of instructions located at arbitrary memory addresses. Therefore, an instruction sequence in a program can contain a gadget that was not present in the program code. Below, we show a binary and assembler codes of a gadget contained in the instruction sequence of the original program [25].

```
f7c7070000000f9545c3 → test edi, 0x7;
            setnz BYTE PTR [ebp-0x3d]
c7070000000f9545c3 → mov DWORD PTR [edi], 0xf000000;
            xchg ebp, eax; inc ebp; ret
```

Gadgets are collected into chains, and their addresses are placed on the stack starting from the return address so that the first gadget transfers control to the second one, the second gadget transfers control to the third one, and so on. Thus, using a chain of gadgets, one can execute malicious code.

Figure 3 shows the state of the stack after placing a ROP chain on it. This chain writes the value `memValue to the address` memAddr. The return address is rewritten by the address of the first gadget. After returning from the function where the overflow occurred, the control is transferred to the first gadget, which loads `memValue` from the stack into the register `eax`. After return (after the execution of the instruction `ret`), the first gadget transfers control to the second gadget, which in turn will load the value of `memAddr` into the register `edx`. Next, the third gadget will save the value of the register `eax` (`memValue`) to the address `edx` (`memAddr`). Next, the control is transferred to the fourth gadget, etc.

Below, we show the same ROP chain in binary form; it writes the value `"/bin"` to the address `0x0830caa0`. An intruder places this sequence of bytes on the stack starting from the return address.

```
00000000  47 65 06 08 2f 62 69 6e  3d 76 07 08 a0 ca 30 08  |Ge../bin=v….0.|
00000010  b5 8b 08 08                                       |…|
00000014
```

3. REVIEW OF EXISTING SOLUTIONS

In this section, we describe the existing solutions to determine the gadget semantics and analyze code-reuse attacks.

3.1. Gadget Semantic Definition

Schwartz et al. [9] proposed to define the gadget semantics by its assignment to types presented in Table 1. The set of gadget types specifies a new instruction set architecture (ISA) in which each gadget type plays the role of instruction. The semantics of each gadget type is defined by a postcondition (Boolean predicate) $\mathcal{B}$ that must always be true after the gadget execution.

It is said that the instruction sequence $\mathcal{I}$ satisfies the postcondition $\mathcal{B}$ if, for any initial state, the postcondition $\mathcal{I}$ is true after the execution of $\mathcal{B}$. The initial state consists of initial register and memory value assignments.

Note that a gadget may be simultaneously assigned to more than one type. For instance, the gadget `push eax; pop ebx; pop ecx; ret` simultaneously copies `eax` to `ebx` and loads a value from the stack into ecx, which corresponds to the types *MoveRegG*: `ebx` ← `eax` and *LoadConstG*: `ecx` ← `[esp + 0]`.

3.1.1. Semantic analysis. To find out if an instruction sequence $\mathcal{I}$ satisfies the postcondition $\mathcal{B}$, Schwartz et al. [9] use the well-known technique of formal verification—computation of the weakest precondition [10]. Put simply, the weakest precondition $\mathrm{wp}(\mathcal{I}, \mathcal{B})$ for the instruction sequence $\mathcal{I}$ and the postcondition $\mathcal{B}$ is the Boolean precondition that describes when $\mathcal{I}$ terminates in the state satisfying $\mathcal{B}$. The weakest preconditions are used to verify that the gadget semantics definition always holds after executing the instruction sequence $\mathcal{I}$. To this end, it is sufficient to verify that

$$wp(\mathcal{I}, \mathcal{B}) \equiv true.$$

If this formula is true, then $\mathcal{B}$ is always true after the execution of $\mathcal{I}$; therefore, $\mathcal{I}$ is a gadget of the semantic type $\mathcal{B}$.

However, the formal verification of gadgets turned out to be very slow in practice. To speed up the process of determining if a gadget can be assigned to one type or another, the gadget instructions are preliminary executed on random input data several times, and the validity of $\mathcal{B}$ is checked. If $\mathcal{B}$ is false after at least one execution, then the instruction sequence cannot be assigned to the type $\mathcal{B}$. Thus, the more complicated computation of the weakest precondition is performed only if $\mathcal{B}$ is true for all executions.

The execution on random input data can also be used to find possible values of the gadget parameters (Table 1). For instance, by examining the values of registers and read memory addresses, it is possible to determine the set of possible values of *Offset* for the *LoadMemG* gadget that loads value to register from memory.

3.2. deRop

In distinction from using the traditional shellcode, which is embedded in the process memory, the return-oriented programming makes it possible to perform arbitrary computations by reusing the code that is already available in the memory. For this reason, it is difficult to use the traditional binary code analysis

Table 1. Gadget types. [*Addr*] denotes memory access at the address *Addr*, ∘ stands for a binary operation, $a \leftarrow b$ means that the final value of a equals the initial value of b, and $X \circ\!\leftarrow Y$ is short for $X \leftarrow X \circ Y$.

Type	Parameters	Definition of semantics
NoOpG	—	Does not change memory or registers
JumpG	AddrReg	IP ← AddrReg
MoveRegG	InReg, OutReg	OutReg ← InReg
LoadConstG	OutReg, Offset	OutReg ← [SP + Offset]
ArithmeticG	InReg1, InReg2, OutReg, °	OutReg ← InReg1 ° InReg2
LoadMemG	AddrReg, OutReg, Offset	OutReg ← [AddrReg + Offset]
StoreMemG	AddrReg, InReg, Offset	[AddrReg + Offset] ← InReg
ArithmeticLoadG	AddrReg, OutReg, Offset, °	OutReg °← [AddrReg + Offset]
ArithmeticStoreG	AddrReg, InReg, Offset, °	[AddrReg + Offset] °← InReg

tools for analyzing ROP attacks. To resolve this difficulty, the tool deRop [11] was proposed. This tool reduces the ROP exploit to the semantically equivalent shellcode, which can be analyzed using the available tools. The authors of [11] mainly use static analysis and identify the following difficulties in the analysis of ROP attacks:

Detection of gadgets. When the stack buffer overflow is exploited, (that will rewrite the return address) a buffer containing arbitrary insignificant data is written on the stack before the address of the first gadget. Moreover, some memory locations may be skipped between the addresses of the first and second gadgets (e.g., if the function in which the overflow occurs clears the arguments from the stack using the instruction `ret n`). Even though deRop tries to use static analysis whenever possible and avoid dynamic analysis, the first two gadgets are detected using a debugger.

Keeping track of the stack pointer. In ROP exploits, the stack pointer is used to obtain the address of the next gadget in the same way as the instruction pointer (program counter) is used to obtain the address of the next instruction. Therefore, in order to detect the next gadget, the stack pointer must be tracked.

Location of the stack and constants. Shellcode usually uses `mov reg, imm` for loading constants into the register, while ROP usually use `pop reg`. The location of the stack in the original ROP chain differs from the stack location in the semantically equivalent shellcode. Therefore, it is required to track the location of constants on the stack.

Function calls. Some gadgets in the ROP chain are used for calling functions. Such calls must be detected and the functions must be called in the conventional way. Moreover, the values of arguments of the function call (including the arguments that are constants or pointers) must be determined.

Loops. ROP chains may contain loops. One must know how to detect them and determine the condition of loop termination.

3.2.1. Postprocessing. As soon as all gadgets have been analyzed, a number of postprocessing steps are performed in order to simplify the output code.

Data in memory. The values of operands of instructions that access memory are computed, and the operands are replaced by constants.

Zero bytes. It is typically required that the shellcode contains no zero bytes because this can lead to truncation of the shellcode after certain operations (e.g., `strcpy`). This difficulty is resolved by replacing all zero bytes by nonzero values and adding a decoder to the beginning of the shellcode that will reconstruct the original values.

Return address. The return address in the exploit is replaced by the address of the beginning of the resulting shellcode.

3.3. ROPMEMU

ROPMEMU [12] is a framework for analyzing complex code-reuse attacks. The authors use the dynamic approach to binary code analysis and distinguish the following problems in the analysis of ROP attacks (C1–C3 have already been mentioned in Subsection 3.2):

[C1] Verbosity. The majority of ROP gadgets contain spurious instructions. For example, the gadget designed for incrementing `eax` can also load (`pop`) a value from the stack before transferring the control to the next gadget (`ret`).

[C2] Stack-based instruction chaining. In distinction from ordinary programs, in which instructions are placed sequentially in the memory, ROP exploits are split into small gadgets that are connected into a chain by indirect control flow instructions (`ret`).

[C3] Lack of values of constants. ROP chains are typically constructed from parameterized gadgets (e.g., load an arbitrary value into the `rax` register) that use parameters stored on the stack.

[C4] Conditional branches. In distinction from the traditional change of the instruction pointer, a branch condition in a ROP chain changes the stack pointer. This means that a simple conditional jump is implemented with a number of gadgets (see [13, pp. 18–19]). To reduce the chain to a more readable code, it is necessary to identify these conditional branches and replace each of them with a single branch instruction.

[C5] Return to functions. Function calls are typically implemented in ROP as simple return (`ret`) to the function's entry point. Since normal gadgets are also often extracted from the code located inside libraries, it is necessary to distinguish a function call from another gadget.

[C6] Dynamically generated chains. A ROP chain does not necessarily completely resides in the memory; rather, gadgets that prepare the execution of other gadgets in future may be used.

[C7] Stop condition. The authors assume that the analyst is able to locate the beginning of a ROP chain in memory. However, the emulation process must be terminated after all the gadgets have been extracted.

The ROPMEMU framework uses a set of various techniques for analyzing ROP chains and for reconstructing the equivalent code in the form that can be analyzed using traditional reverse engineering tools, such as IDA Pro [14]. It is assumed that the analyst has at his disposal the memory dump and the entry point of (the first) ROP chain in it. The other dynamically generated chains are reconstructed by the framework, which has five main phases of analysis.

3.3.1. Multipath emulation. At this step, the assembler instructions used by the ROP chain (C2) are emulated. All possible branches are examined, and for each execution path an independent trace (annotated with the values of registers and memory) is generated.

The emulator also detects returns into library functions, skips their body, and simulates their execution by generating spurious data and return value (C5).

The emulator first reads the memory content from the dump and maintains the shadow memory [15]. The stop condition (C7) is determined by a set of heuristics based on the locality principle (the emulator detects a large relative increment of the stack pointer) and the gadget length not taking into account the detected function calls. As soon as the stop condition holds true, the contents of the shadow memory and the execution trace are saved to disk and are explored to detect new ROP chains. If new ROP chains are detected, then the emulator is restarted to analyze the next chain and so on until all dynamically generated chains are detected and analyzed (C6).

For ROP chains with a complex control flow, the simple approach based on emulation is insufficient for the analysis of the entire ROP exploit. Indeed, the coverage is limited by the executed branches only, which often depend on the dummy return values of functions generated by the emulator. This problem can be resolved using multipath emulation, which is a version of the multipath execution algorithm [16] adapted to ROP chains. In particular, the emulator recognizes the situation in which the stack pointer is modified depending on the contents of the flag register. At the end of the emulation process, the list of all branch points together with the flag values at each of them is obtained. The emulator is then restarted with the instruction to take another path at the branch point. Thus, the execution follows another path. The exploration of branches is terminated when all branches have been explored.

However, in presence of loops in the ROP chain, the emulator could get trapped in an endless execution path. The solution in this case is to keep track of the number of occurrences of the stack pointer during the execution of branch-related instructions. If this number is above a certain threshold, the emulator inverts the control transfer to terminate the loop and explore the remaining part of the control flow graph.

3.3.2. Trace splitting. In this phase, traces generated by the emulator are analyzed, duplicates are removed, and unique blocks of code are extracted. Each trace is cut at each branch point, and a new block is generated and saved to a separate trace. As a result, a set of traces associated with each "basic block" in the chain is produced.

3.3.3. Unchaining. In this phase, all unconditional jump instructions (`ret, call, jmp`) are removed from the trace, and the contents of the sequentially executed gadgets are joined into a single basic block (C2). Then, the `mov` instructions are simplified by computing their operands (e.g., `mov rax, [rsp + 0x30]`). The `pop` instructions are replaced by `mov` instructions, and the required values are fetched from the corresponding locations on the stack (C3).

3.3.4. Control flow graph recovery. In this phase, all traces are joined into a unified graph representation. Then, this graph is translated into an x86 program due to recognizing instructions associated with conditional branches and replacing them by traditional instructions that use conditional jumps that use the instruction pointer (C4).

The next task of this phase is to detect and re-roll loops. ROP chains can contain return-oriented loops and unrolled loops. In the first case, ROP instructions are used to repeatedly execute the same block of gadgets on the stack with conditional exit. The unrolled loops repeat the same sequence of gadgets for a predetermined number of times. The framework automatically detects recurrent patterns and replaces them by a more compact piece of assembler code that is a semantically equivalent loop.

The resulting code is wrapped within a valid function prologue and epilogue and then included in a separate ELF file to enable one to explore it using traditional reverse engineering tools (such as IDA Pro [14]).

3.3.5. Binary optimization. The final step of the analysis consists of applying standard compiler transformations to simplify the generated assembler code in the ELF file. In particular, dead code is removed, the transformations described in Subsection 3.2.1 are applied, and a purified and optimized version of the exploit is generated (C1).

4. METHOD FOR ANALYZING CODE-REUSE ATTACKS

The method for analyzing code-reuse attacks proposed in this paper makes it possible to reconstruct the semantics of a ROP chain and track the progress of attack. Given the binary ROP chain, the sequence of called gadgets is reconstructed. The found gadgets are classified under semantic types and their parameter values are determined. In addition, invocations of functions and system calls are detected in the chain, and their prototypes and values of arguments are reconstructed. Note that in this paper our goal is to explore at least one execution path of the ROP chain rather than all execution paths. For this reason, the proposed method does not take into account the conditional branches in ROP chains.

4.1. Gadget Frame

To decompose a binary ROP chain into gadgets, we introduce the concept of gadget frame, which is similar to the stack frame in x86. The chain of gadgets is decomposed into frames. The gadget frame contains the values of the gadget parameters (e.g., the value loaded into a register from the stack by the gadget *LoadConstG*) and the address of the next gadget. The beginning of the frame is determined by the value of

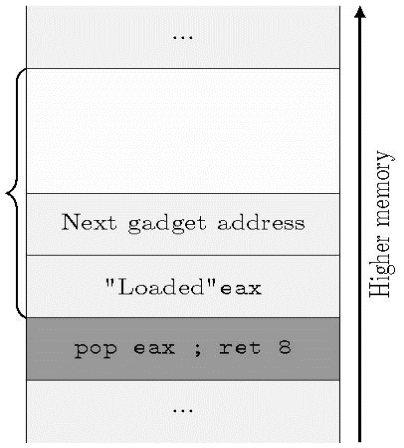

Fig.4. `pop eax ; ret 8` gadget frame

the stack pointer before the execution of the first gadget instruction.

In Fig. 4, the brace shows the boundaries of the gadget frame (`pop eax ; ret 8`). This gadget loads a value from the stack into `eax`, which corresponds to the gadget type that loads a constant *LoadConstG*: `eax ← [esp + 0]`. The size of the gadget frame is `FrameSize = 16`, and the address of the next gadget is at offset 4 from the beginning of the frame (`NextAddr = [esp + 4]`).

4.2. Classification of Gadgets

The classification of gadgets proposed in [17] makes it possible to determine their semantics. The gadget semantics is defined by a set of Boolean postconditions (gadget types), satisfied by the gadget's instructions, and the values of the postconditions' parameters (Subsection 3.1). The set of gadget types proposed by Schwartz et al. [8] (Table 1) turned out to be insufficient for analyzing ROP chains found on the Internet; so, it was expanded by additional types presented in Table 2. Furthermore, we added gadget types that do not guarantee the control preservation (at the bottom of Table 2).

A gadget is classified on the basis of the effects discovered by its execution on random input data. The gadget instructions are translated into an intermediate representation. Then, the interpretation of the intermediate representation is started. During the interpretation, register and memory accesses are tracked. If a register or memory region is read for the first time, then the obtained value is generated randomly. As a result of interpretation, the initial and final values of registers and memory are obtained. Based on these data, the gadget is tentatively assigned to a certain type. For example, to be assigned to the type *MoveRegG*, a pair of registers such that the initial value of the first register equals the final value of the second

Table 2. Extended gadget types. [*Addr*] denotes memory access at the address *Addr*, $\circ$ stands for a binary operation, $a \leftarrow b$ means that the final value of a equals the initial value of b, and $X \circ\!\leftarrow Y$ is short for $X \leftarrow X \circ Y$.

Type	Parameters	Definition of semantics
JumpMemG	AddrReg, Offset	IP ← [AddrReg + Offset]
GetSPG	OutReg	OutReg ← SP
InitConstG	OutReg, Value	OutReg ← Value
InitMemG	AddrReg, Value, Offset, Size	[AddrReg + Offset] ← Value
NegG	InReg, OutReg	OutReg ← −InReg
ArithmeticConstG	InReg, OutReg, Value, $\circ$ (+/⊕)	OutReg ← InReg $\circ$ Value
ShiftStackG	Offset, $\circ$ (+/−)	SP $\circ\!\leftarrow$ Offset
PushAllG	—	([ESP − 4] ← EAX) ∧ ([ESP − 8] ← ECX) ∧ ([ESP − 12] ← EDX) ∧ ([ESP − 16] ← EBX) ∧ ([ESP − 20]← ESP) ∧ ([ESP − 24] ← EBP) ∧ ([ESP − 28] ← ESI) ∧ (EIP ← EDI)
Do not preserve control		
JumpSPG	—	IP ← SP
CallG	AddrReg	IP ← AddrReg
CallMemG	AddrReg, Offset	IP ← [AddrReg + Offset]
IntG	Value	Invoke interrupt Value
SyscallG	—	System call

register must exist. As a result of the analysis, a list of all types and their parameters to which the gadget can be assigned is composed (the list of candidates). Then, the interpretation is run several more times with different input data, and the incorrectly determined types are removed from the list.

As a result of the gadget classification, the gadget semantic types and their parameters, as well as information about the gadget frame (Subsection 4.1), i.e., the frame size (FrameSize) and the offset of the location containing the address of the next gadget relative to the beginning of the frame (NextAddr), are obtained.

Note that a gadget is classified as a result of its execution on a limited set of inputs, which generally does not guarantee that the semantics will hold when the gadget is executed on arbitrary input data. For the exact classification, the formal verification of the gadget semantics must be performed as described in Subsection 3.1.1. Therefore, a gadget can be classified incorrectly. However, the number of the incorrectly classified gadgets after execution on 10 random inputs is insignificant and is acceptable for reconstructing the semantics of ROP chains.

4.3. Recovery of ROP Chain Semantics

The binary ROP chain is loaded on the shadow stack. Using the information about the preceding gadget frame obtained as a result of the classification, the gadgets in the chain are classified one after another. The next gadget address offset relative to the beginning of the frame and the frame size actually show where the address of the next gadget to be classified should be taken and where its frame begins, respectively. The pointer of the shadow stack always points to the beginning of the last classified gadget frame.

To reconstruct the values of registers and memory before the gadget execution (e.g., to reconstruct the arguments of the system or function call) we maintain the shadow memory [15] that is common for all gadgets. Initially, the shadow memory is empty. Successively, for each classified gadget of the chain, the interpretation process of its intermediate representation with the shadow memory is run several times; the shadow memory acts as the initial data of the register and memory values. The read registers and memory bytes, which are not contained in the shadow memory, are generated randomly at each interpreter run. The final values of the registers and memory that were the same for all runs are updated in the shadow memory.

The values of all constants loaded by the ROP chain can be recovered from the shadow stack. For this purpose, the gadgets classification is insufficient because it does not take into account the data on the shadow stack but rather randomly generates the data read from the stack. The classification of *LoadConstG* gadget makes it possible to determine the register *Out-*

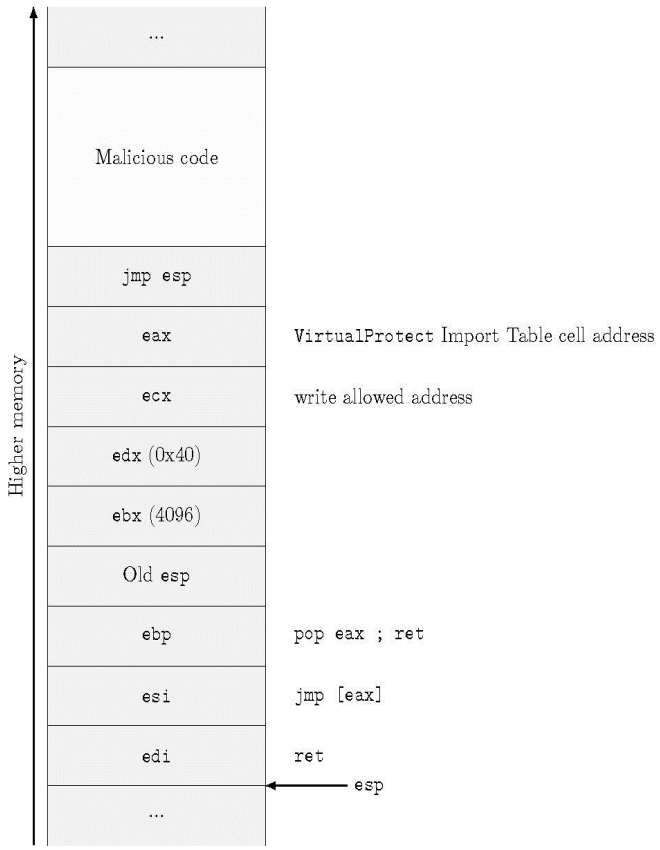

Fig.5. State of the stack after executing the `pushad` instruction.

Reg into which the constant is loaded and the *Offset* at which the constant is read from the stack. After the classification of *LoadConstG*, *OutReg* value is loaded from the shadow stack at the *Offset* from the shadow stack pointer and added to the shadow memory.

In order to bypass DEP in 32-bit Windows programs, *PushAllG* (`pushad; ret`) is often used to call the WinAPI function `VirtualProtect` [18] (which makes the stack executable), then the control is transferred to the shellcode residing up the stack (Fig. 5). The point is that the instruction `pushad` saves the general-purpose registers on the stack. If these registers are preliminary initialized with proper values, then the stack will contain an ordinary ROP chain.

Firstl, the address of *NoOpG* (`ret`) is loaded into the register `edi`, and the address of the gadget that should call the function `VirtualProtect` is loaded into `esi` (e.g., `jmp [eax]`; in this case, the address of the `VirtualProtect` cell in the table of imported symbols is first loaded into `eax`). The VirtualProtect arguments number 2–4 are loaded into the registers `ebx`, `edx`, and `ecx`, respectively. The address of *ShiftStackG* (`pop eax; ret`), which increments the stack pointer, acts as the return address from VirtualProtect and is loaded into the register `ebp`. After executing the `pushad` instruction, the values of these

registers will be on the stack, as shown in Fig 5. In turn, the execution of the instruction ret transfers control to the address of the gadget written to the last saved register `edi` (`ret`). Next, the control goes to the gadget that calls `VirtualProtect(esp, ebx, edx, ecx)`. After return from `VirtualProtect`, the stack becomes executable, and the control goes to the gadget which address was earlier loaded into the register `ebp` (`pop eax; ret`). As a result, the gadget *JumpSPG* (`jmp esp`) will be called, which transfers control to an ordinary shellcode placed immediately up the stack, which is now executable.

After the classification of *PushAllG*, the corresponding values of the registers are written on the shadow stack. In turn, the gadget *JumpSPG* is interpreted as the transfer of control to the ordinary shellcode residing on the stack, and its bytes are disassembled.

Note that a ROP chain can preliminary write a gadget to memory in order to use it later. We demonstrate an example chain below. First, the machine code of the gadget `mov [eax + ebp * 4], ebx; ret` is loaded into the register `edx`. Then, this gadget is saved to the memory address `eax`. Next, the gadget parameters are loaded: `ebx` and `ebp`. Finally, the control is transferred to the address `eax`, where resides the preliminary saved gadget that writes the value of the register `ebx` to the memory at the address `eax + ebp * 4`.

```
pop edx; ret  // edx = "\x89\x1c\xa8\xc3"
mov [eax], edx; ret
pop ebx; pop ebp; ret
jmp eax  // mov [eax + ebp * 4], ebx; ret
```

Thus, if during analysis of a ROP chain, the address of the next gadget is in the shadow memory, then the gadget from the shadow memory is classified. If it turns out after classification that this gadget cannot be assigned to any type, then it is assumed that this is the transfer of control to the ordinary shellcode that were earlier saved to memory. The bytes of the shellcode in the shadow memory are also disassembled.

4.3.1. Reconstruction of functions and system calls. A function can be called from a ROP chain using the gadgets JumpG, JumpMemG, CallG, and CallMemG, or its address can be simply placed on the stack. The system call is made by the gadget IntG in the 32-bit operating system and by SyscallG in the 64-bit operating system. The system call number and the values of function and system call arguments are recovered from the shadow memory. If a null-terminated string resides at the argument address in the shadow memory, then it is also recovered.

For ROP chains under Linux, the name of the system call is found by its number. The called function name can be recovered if the function was called at the address that was read from the imported symbol table (GOT in ELF and IAT in PE). In Linux, the prototypes of functions and system calls are sought in man-pages [19]; the prototypes of functions in Windows are sought using the API Monitor [20].

5. SOFTWARE IMPLEMENTATION

We implemented the method described above as a software tool. This tool gets at its input a binary ROP chain and the executable file containing the gadgets used in this chain. Note that in this paper we do not seek the ROP chain in the exploit—this task should be accomplished by an analyst. We support the following formats of the executable files: ELF32, ELF64, PE32, and PE32+. Chains that use gadgets from different executable files are currently not supported.

5.1. Interpretation of the Intermediate Representation of Gadget Instructions

In this paper, we use the intermediate representation of instructions developed in the Institute for System Programming, Russian Academy of Sciences [21], which is in the SSA-form and has a three-address code. The memory and register address spaces are represented by two byte arrays. The register address space consists of all machine registers with account for their overlapping and intersections. To take into account side effects, the status word similar to the flag register in x86 is used.

The gadget instructions are translated into the intermediate representation the interpretation of which gives the initial and final values of the registers and memory. Initially, for all address spaces, the maps of loaded and stored values are empty. The instructions of the intermediate representation are replaced by the equivalent blocks of x86-64 instructions. The *STORE* instructions are replaced by the calls of the function that updates the map of stored values. The *LOAD* instructions are replaced by the calls of the function that returns the current value. This function performs one of the following actions:

- reads the value from the map of stored values if the value is in the map;
- reads the value from the map of loaded values if it is in the map of loaded values and not in the map of stored values;
- inserts a randomly generated value in the map of loaded values if the first access to given address occurs.

Table 3. The list of analyzed ROP exploits

Application	CVE number	Platform	Gadgets from
MongoDB	CVE-2013-1892	Linux x86	mongod
Nagios3	CVE-2012-6096	Linux x86	history.cgi
ProFTPd	CVE-2010-4221	Linux x86	proftpd
Nginx	CVE-2013-2028	Linux x64	nginx
AbsoluteFTP	CVE-2011-5164	Windows x86	MFC42.dll
ComSndFTP	N/A 2012-06-08	Windows x86	msvcrt.dll

Then, the resulting x86-64 code is executed. As a result, the initial and final states of the address spaces are obtained.

5.2. Analyzing the ROP Chain

The tool emulates loading of the executable file into the virtual address space with all relocations applied. In the loaded executable file, the instructions of each gadget at its address are disassembled up to the control transfer instruction. The instructions thus obtained are translated into the intermediate representation and are classified. The arguments of functions and system calls are recovered according to the calling convention from the shadow memory; as the return value of the function, a dummy value is added to the shadow memory. Then, the shadow stack and the shadow memory are updated as described in Subsection 4.3.

As a result, the tool produces a text file with all sequentially called gadgets and their types and parameters listed. Moreover, the resulting file contains the prototypes of the invoked functions and system calls with their recovered argument values. If a ROP exploit ends with the call of an ordinary shellcode, then its disassembled instructions are also written to this file.

6. PRACTICAL RESULTS

We successfully used the method for analyzing code-reuse attacks proposed in this paper to analyze real-world ROP exploits found on the Internet. We extracted the binary ROP chains by hand. Then, we detected the executable file containing the gadgets used in the chain.

Good starting points for finding ROP exploits were the penetration testing framework Metasploit [22] and the open exploit database EDB [23]. Unfortunately, the executable file from which gadgets were gathered into a chain is rarely attached to the exploit. At best, the version of the program, the operating system, and (or) distribution are indicated. For this reason, one often has to seek the executable files by hand and check if the gadgets addresses used in the exploit point to the same assembly instructions as provided in the exploit comments. To find old versions of Debian packages, there is a useful project snapshot.debian.org [24], which saves the current state of the Debian distribution several times a day, which significantly simplifies the search of the old versions.

Table 3 contains the list of ROP exploits that were successfully analyzed using the developed ROP chain analysis tool. The time taken by the analysis did not exceed a couple of seconds.

7. CONCLUSIONS

We proposed a method for analyzing code-reuse attacks and implemented it as a software tool. This method simplifies for an analyst the reverse engineering of ROP exploits. Given a binary ROP chain, the list of called gadgets is recovered, and their functionality is semantically described in terms of a Boolean predicate that must be always true after the gadget execution. Moreover, the prototypes and values of arguments of the called functions and system calls are reconstructed. Thus, an analyst can get a view of the ROP chain semantics. We successfully used the implemented method to analyze real-world ROP exploits found on the Internet.

The method is based on the dynamic interpretation of an intermediate representation of the ROP chain instructions. The gadget semantics is recovered by analyzing the effects of the gadget execution on various random input data. To reconstruct the values of functions and system calls arguments, we support a shadow memory in the course of analysis.

A promising direction of future research is the support of the analysis of conditional branches and loops in ROP chains. To improve the accuracy of recovering the gadget semantics, one can use various formal verification techniques. A technical task is the support of the analysis of ROP chains that use gadgets from a number of executable files simultaneously.

FUNDING

This work was supported by the Russian Foundation for Basic Research, project no. 17-01-00600.

REFERENCES

1. Belevantsev, A. and Avetisyan, A., Multi-level static analysis for finding error patterns and defects in source code, 2018. https://doi.org/10.1007/978-3-319-74313-4_3
2. Gerasimov, A.Y., Directed dynamic symbolic execution for static analysis warnings confirmation, *Program. Comput. Software,* 2018, vol. **44**, no. 5, pp. 316-323. https://doi.org/10.1134/S036176881805002X
3. Gerasimov, A. and Kruglov, L., Reachability confirmation of statically detected defects using dynamic analysis, *Proc. of the 11th International Conference on Computer Science and Information Technologies, CSIT 2017,* pp. 60-64. https://doi.org/10.1109/CSITechnol.2017.8312141
4. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org
5. Vulnerabilities (CVE) by year. https://www.cvedetails.com/browse-by-date.php
6. Nurmukhametov, A.R., Zhabotinskiy, E.A., Kurmangaleev, S.F., Gaissaryan, S.S., Vishnyakov, A.V., Fine-Grained Address Space Layout Randomization on Program Load, *Program. Comput. Software,* 2018, vol. **44**, no. 5, pp. 363-370. https://doi.org/10.1134/S0361768818050080
7. CWE-121: Stack-based Buffer Overflow. https://cwe.mitre.org/data/definitions/121.html
8. Shacham, H., "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. of the 14th ACM Conf. on Computer and Communications Security,* 2007, pp. 552–561.
9. Schwartz, E.J, Avgerinos, T., and Brumley, D. Q: Exploit hardening made easy, *Proc. of the 20th USENIX Conference on Security, SEC'11,* USENIX Association, 2011, p. 25.
10. Jager, I. and Brumley, D., Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, 2010.
11. Lu, K., Zou, D., Wen, W., and Gao, D., deRop: Removing return-oriented programming from malware, *Proc. of the 27th Annual Computer Security Applications Conference, ACSAC'11,* ACM, 2011, pp. 363–372.
12. Graziano, M., Balzarotti, D., and Zidouemba, A., ROPMEMU: A framework for the analysis of complex code-reuse attacks, *Proc. of the 11th ACM on Asia Conference on Computer and Communications Security, ASIA CCS'16,* ACM, 2016, pp. 47–58.
13. Roemer, R., Bbuchanan, E., Shacham, H., and Savage, S., "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.,* 2012, vol. 15, no. 1, pp. 2:1–2:34.
14. IDA Pro. https://www.hex-rays.com/products/ida/
15. Nethercote, N. and Seward, J., How to shadow every byte of memory used by a program, *Proc. of the 3rd International Conference on Virtual Execution Environments, VEE'07,* ACM, 2007, pp. 65–74.
16. Moser, A., Kruegel, C., and Kirda, E., Exploring multiple execution paths for malware analysis, *Proc. of the 2007 IEEE Symposium on Security and Privacy, SP'07,* IEEE Computer Society, 2007, pp. 231–245.
17. Vishnyakov, A.V., Classification of ROP gadgets, *Trudy ISP RAN,* 2016, vol. **28**, no. 6, pp. 27–36.
18. VirtualProtect function (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/aa366898(v=vs.85).aspx
19. The Linux man-pages project. https://www.kernel.org/doc/man-pages/
20. API Monitor: Spy on API Calls and COM Interfaces. http://www.rohitab.com/apimonitor
21. Padaryan, V.A., Soloviev, M.A., and Kononov, A.I., Modeling operational semantics of machine instructions, *Trudy ISP RAN,* 2010, vol. **19**, pp. 165–186.
22. Metasploit Framework. https://github.com/rapid7/metasploit-framework
23. Exploit Database. https://www.exploit-db.com
24. snapshot.debian.org. http://snapshot.debian.org
25. Salwan, J., An introduction to the Return Oriented Programming and ROP-chain generation, 2014. http://shell-storm.org/talks/ROP_course_lecture_-jonathan_salwan_2014.pdf

Translated by A. Klimontovich

SPELL; OK