ISSN 0361-7688, Programming and Computer Software, 2021, Vol. 47, No. 4, pp. 271–297. © Pleiades Publishing, Ltd., 2021.

Survey of Methods for Automated Code-Reuse Exploit Generation

A. V. Vishnyakov[a],* and A. R. Nurmukhametov[a],**

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

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

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

Received November 11, 2020; revised December 5, 2020; accepted December 20, 2020

Abstract—This paper provides a survey of methods and tools for automated code-reuse exploit generation. Such exploits use code that is already contained in a vulnerable program. The code-reuse approach allows one to exploit vulnerabilities in the presence of operating system protection that prohibits data memory execution. This paper contains a description of various code-reuse methods: return-to-libc attack, return-oriented programming, jump-oriented programming, and others. We define fundamental terms: gadget, gadget frame, gadget catalog. Moreover, we show that, in fact, a gadget is an instruction, and a set of gadgets defines a virtual machine. We can reduce an exploit creation problem to code generation for this virtual machine. Each particular executable file defines a virtual machine instruction set. We provide a survey of methods for gadgets searching and determining their semantics (creating a gadget catalog). These methods allow one to get the virtual machine instruction set. If a set of gadgets is Turing-complete, then a compiler can use a gadget catalog as a target architecture. However, some instructions can be absent. Hence we discuss several approaches to replace missing instructions with multiple gadgets. An exploit generation tool can chain gadgets by pattern searching (regular expressions) or considering gadgets semantics. Furthermore, some chaining methods use genetic algorithms, while others use SMT-solvers. We compare existing open-source tools and propose a testing system rop-benchmark that can be used to verify whether a generated chain successfully opens a shell.

DOI: 10.1134/S0361768821040071

1. INTRODUCTION

Modern software is known to contain bugs. Some researchers consider such errors as inevitable. However, not all errors can be used to harm. Exploitable errors are called vulnerabilities. When exploited, the vulnerability causes serious consequences such as money losses, degradations of communication, compromise of cryptographic keys [1]. With the development of the Internet of things, one can exploit things that surround us daily, such as kettles, refrigerators, and shower systems. Medical equipment safety issues are crucial. Halperin et al. [2] showed that it is possible to exploit implanted heart defibrillators.

Along with the security development lifecycle, methods to detect various software defects are also improving. In response to the improvement of protection methods against vulnerabilities exploitation, new methods are being developed to bypass and exploit them. Hence, it is necessary to know and understand the principles of both software protection and attacks. Moreover, vendors and software developers can require a proof of concept (exploit) to prioritize a vulnerability fix.

The stack buffer overflow is likely to be one of the most exploited software defects [3] because it is easy to use it for the control-flow hijacking. In simplest case of a total lack of protection, exploitation goes as follows. The return address located on stack above the local buffer is overwritten with a controlled value. This value points back to the buffer that contains the code that the attacker wants to execute.

The DEP protection has appeared to counter the code execution on stack. DEP prohibits writing to code regions and execution in stack and heap process memory. This protection puts an end to the code injection into process memory. The attackers were restricted in executing just the code available in process memory. In response to DEP ubiquity, code-reuse attacks began to be developed rapidly. The first one was a return-to-libc attack [4]. The return address is replaced by the address of the function to be called, followed by its arguments. Return-oriented programming (ROP) [5–7] is a generalization of this technique. In return-oriented programming, one uses gadgets instead of functions. A *gadget* is a short instruction sequence ending with a return instruction. Gadgets are chained together so that they sequentially transfer control to one another and carry out a malicious payload. Shacham [5] defined the term gadget and introduced the first gadget catalog for the x86 instruction set. He also proved that this catalog is Turing-complete. After that, the applicability of ROP was also shown for other architectures: ARM [8–12], SPARC [13], Atmel AVR [14], PowerPC [15], Z80 [16], MIPS [15]. Papers [9, 17–19] showed that gadgets ending with not only return instructions could be used.

ROP can be used as a steganographic method [20]. Ntantogian et al. [21] proposed using code-reuse methods to hide malicious functionality from detecting by antivirus tools, while Mu et al. [22] suggested using ROP to obfuscate the code. Code-reuse methods allow for backdoors to be inserted into software [23, 24].

With the development of the code-reuse methods, tools were also being developed, helping the attacker construct such attacks. At first, this process was almost manual, but over time it gradually became automated. At the moment, the literature presents a set of approaches to automated construction of code-reuse exploits [6, 7, 11–13, 18, 25–31]. For some of them, even the tools are available [32–44].

This work aims to provide a detailed study of available methods and tools for automated code-reuse exploit generation to determine their strengths and weaknesses and identify future directions for the research.

In addition to practical use, methods and tools considered in the paper may be of scientific interest. The task of automated exploit generation is to translate some exploit description into code for the virtual machine instruction set architecture, implicitly set by the memory state of the exploitable process. Gadgets in the process memory are like instructions. What is more, the exploited executable file provides an instruction set that is not known a priori. To learn this instruction set, one needs to find all the gadgets and determine their functionality (semantics). As a result, one creates a catalog of gadgets that describes their semantics. A gadget catalog is an input data for the tool that generates exploits. The exploit generation tool should consider that a set of gadgets, unlike processor instructions, may lack some instructions, while others may have non-trivial side effects. It complicates the development of the tools for automated generation of ROP exploits.

The paper has the following structure. Sections 2–10 provide a survey of attacks and defense mechanisms. Section 11 describes a general scheme for code-reuse exploit generation. Section 12 introduces a definition of the *gadget catalog*. Section 13 describes approaches to finding gadgets. Section 14 provides methods to determine the gadget semantics. Section 15 reviews methods for gadget chains generation. Section 16 reveals the problem of accounting restricted symbols in chains. Section 17 presents an experimental comparison of open-source tools done by the specially developed rop-benchmark testing system [45]. The last Section 18 discusses the problems of the existing methods and identifies further research directions.

2. BACKGROUND

2.1. Data Execution Prevention

Data Execution Prevention is an operating system protection that prohibits execution of memory pages marked as “data”. A memory page can be simultaneously accessible either for writing or for execution, but not both, which precisely reflected in the name of the OpenBSD WX security policy [46] (**W**rite XOR e**X**ecute). On Windows, this defense mechanism is called DEP (Data Execution Prevention) [47]. Linux [48] and Mac OS X have similar protective mechanisms. The protective mechanism is implemented in the hardware using a special NX-bit (**N**o e**X**ecute), which marks the pages inaccessible for execution. If the processor lacks the hardware support for the NX-bit, then that mechanism is emulated by the software.

In classic exploitation of stack buffer overflow [49], the attacker injects a malicious code into the buffer and transfers the control onto that code. The protection does not allow one to execute the injected code as its location is the stack marked as “data”.

2.2. Code-Reuse Attacks

Code-reuse attacks have appeared to bypass the protection that prevents data execution. The idea is not to inject a malicious code but to reuse the code already presented in the program and libraries to implement the functionality of the malicious code. The stack buffer overflow vulnerability or the ability to write an arbitrary value to an arbitrary memory location (write-what-where [50]) allows one to replace the return address with the address of some code from the program address space. Thus, after returning from the function, the control is transferred to this code.

2.3. Return-to-Library Attack

Alexander Peslyak was the first to show that exploitation is possible even with a non-executable stack and proposed a return to library attack (return-to-libc) [4]. The attacker substitutes the return address with some library function address and places its arguments up the stack. For example, the attacker may call `system(“/bin/sh”)` from the standard library `libc`. Thus, the attacker may open the operating system shell.

2.4. Address Space Layout Randomization

An address space layout randomization (ASLR) [51] is a protective mechanism of the operating system

Table 1. Example of x86 gadgets

`mov eax, ebx ; ret`	Copying `ebx` register value into `eax` register.
`pop ecx ; ret`	Loading the value from stack on `ecx` register.
`add eax, ebx ; ret`	Adding `ebx` register value to `eax` register.

that loads memory segments at different base addresses for each program run. This protection makes it difficult to conduct a return-to-library attack (return-to-libc), as the base address of the library `libc` is random, and the `system` function address is unknown before the program loading. However, for compatibility with ASLR, the program must be compiled into a position-independent code [52], which is not always held. For example, in Linux, the base addresses of dynamic libraries, stack and heap are randomized, while the base address of the program image often remains constant [53].

If the library base address is random, but the program image is not, then the attacker can call the imported function through the procedure linkage table PLT [54], which contains a code for calling library functions. The return-to-plt attack is a modification of the return-to-library attack and consists of replacing the return address with the address of the code from PLT that calls the function from the dynamic library.

3. RETURN-ORIENTED PROGRAMMING

Shacham [5] suggested the term return-oriented programming (ROP). ROP is an effective method to bypass data execution prevention (DEP [47], WX [46]). In a certain sense, this approach is the generalization of the return-to-library attack. However, the malicious payload is implemented not by calling one function, but is formed from several code pieces already present in the program, which are called *gadgets*. A gadget is an instruction sequence ending with a control transfer instruction. Each gadget modifies the state of registers and memory. For instance, it adds the values of two registers and writes the result to the third one. Having studied all the gadgets available in the program, the attacker links them into chains in which the gadgets sequentially transfer the control to each other. Such a chain of gadgets carries out the total malicious payload. With a sufficient number of gadgets, the attacker can form a Turing-complete set to allow arbitrary computation [5]. It is worth noticing that ROP is also useful when partial randomization of the address space is present. In this case, gadgets from non-randomized memory areas are used.

For clarity, Table 1 demonstrates the assembly code[1] of the three gadgets for x86. Each of the gadgets ends with a `ret` instruction, which allows transferring control to the next gadget via the address placed on stack.

The x86 architecture is CISC. The x86 instructions are not fixed in length, and each instruction can execute several other low-level commands. The number of commands is enormous, and their encoding is so tight that almost any sequence of bytes is the correct instruction. Besides, due to the different command lengths (from 1 byte to 15), the x86 architecture does not require instructions alignment. From the ROP point of view, it means the following. The set of gadgets in the program is not limited only to compiler-generated instructions. It enlarges with the instructions not presented in the original program but received upon access to the middle of other commands. Here is an illustrating example [55]:

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

An executable file essentially defines the set of gadgets that can be used to compose a ROP chain. Furthermore, for another executable file, the ROP chain has to be reassembled anew. The ROP chain can be considered as a program for some virtual machine defined by an executable file [56]. The stack pointer acts as a program counter for this virtual machine. The operation codes (gadget addresses) and their operands are located on stack. Graziano et al. [57] even proposed a tool for translating ROP chains into a regular x86 program. Figure 1 shows an example of a gadget chain located on stack that stores `memValue` to `memAddr`. Opcodes (gadget addresses) are located from the return address on stack and are shaded in dark gray. The operands `memValue` and `memAddr` are shaded in light gray. Curly brackets denote virtual machine instructions (opcode and its operands). The actual gadget instructions on x86 are given on the right. At the beginning of the chain, where in normal execution the function return address is located, we place the opcode – the address of the first gadget. Then the `memValue` operand is located, which the

[1] Hereafter, we will use Intel syntax for x86 assembler.

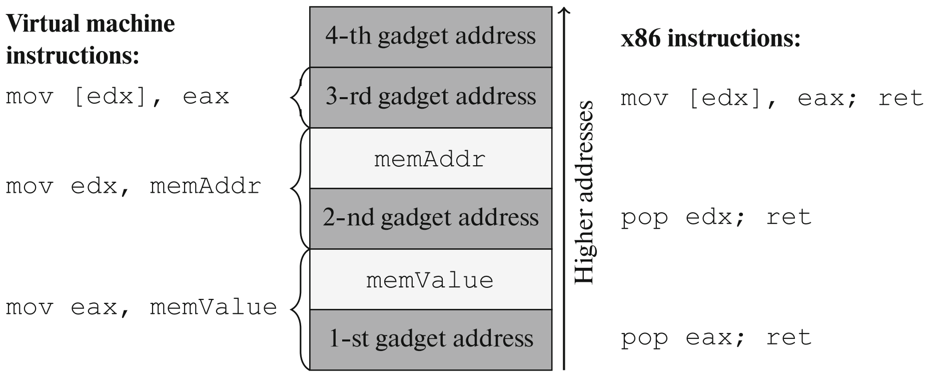


Fig. 1. A ROP Chain, storing `memValue` to `memAddr`.

first gadget loads into the `eax` register. Then follows the address of the second gadget to which the first gadget transfers control via the ret instruction, and so on.

Return instructions on x86 are encoded as: `c3`, `c2**`, `cb`, `ca**` (where any bytes can be instead of stars). Encoding the return instruction in such a way results in many gadgets in x86 code. Even relatively small binary files contain gadgets that are practically applicable from the attackers' point of view. Schwartz et al. [6, 58] provide statistics that among the programs larger than 100KB, about 80% contain sets of gadgets that allow one to call any function from the library which is dynamically linked with a vulnerable application.

Subsequently, the use of ROP was successfully demonstrated for other architectures: SPARC [13], Z80 [16] (Harvard architecture voting machine of the late 80s), ARM [8–12, 59]. The above-mentioned works showed that on RISC architectures it was possible to construct a set of gadgets both serviceable and Turing-complete. RISC architectures are often characterized by fixed commands length, requirement to align instructions on their size, and simplified access to memory (only store and load instructions access to memory). The instructions alignment, compared to x86, forces attacker to find gadget that the program code originally contained. These gadgets are usually valid function epilogues.

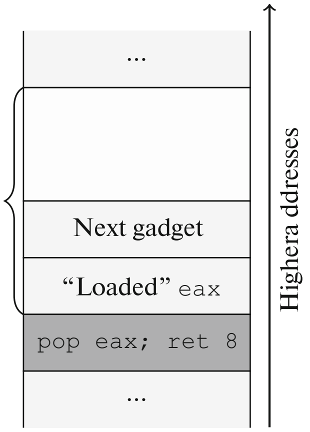


Fig. 2. `pop eax ; ret 8` gadget frame.

3.1. Gadget Frame

In order to place a ROP chain on stack, it is convenient to introduce the concept of a *gadget frame* [60] similar to the x86 stack frame. A chain of gadgets is assembled from the frames. The gadget frame contains values of gadget parameters (for instance, the value loaded onto the register from the stack) and the address of the next gadget. The beginning of the frame is determined by the value of the stack pointer before executing the first gadget instruction. In Fig. 2, curly brackets denote the borders of the pop eax; ret 8 gadget frame. The gadget loads the value from the stack into `eax` at offset 0 from the beginning of the frame. The size of the gadget frame is `FrameSize = 16`, and the next gadget address is located at offset 4 from the beginning of the frame (`NextAddr = [esp + 4]`).

3.2. Returning to Randomized Library Attack

Roglia et al. [61] showed how to call a function from the library with ROP, even though the library base address is randomized (the vulnerable program image base considered not to be randomized). Linux stores the addresses of imported functions in the section `.plt.got` [54] to perform dynamic linkage (Windows has a similar mechanism via the Import Address Table [62]). The attacker can use this information to calculate the addresses of the remaining functions from dynamically linked libraries. Suppose that `.plt.got` contains the address of the imported `open` function from `libc`. Then the address of function `system` can be calculated with the following formula:

$$system = open + (offset(system) - offset(open)),$$

where *offset*(*s*) function returns the offset of *s* function relatively to the library base address. ASLR randomizes the base address of the library load, while the offset value of `system` function relative to `open` inside the library (*offset*(*system*) – *offset*(*open*)) remains constant and is known to the attacker in advance.

The attacker creates a ROP chain which loads the address of `open` function from `.plt.got`, add the previously known offset of `system` relative to `open` to the loaded address, and transfer the control to the calculated address, i.e., to `system` function. The attacker can also make a chain that adds to the address of `open` function a necessary offset in the `.plt.got` memory and call the function by its address from that memory. If it is necessary to call the imported function, then a ROP chain can be made up that calls the function by its address from `.plt.got`, or one can perform the return-to-plt attack (Section 4), where the code in PLT calls this function.

It is worth noticing that Linux uses the lazy binding mechanism. Initially, the addresses of the stub functions are written in `.plt.got` instead of the addresses of the imported functions. When the imported function is called for the first time, the stub function dynamically binds it and write its virtual address in `.plt.got`. Thus, the address of `system` function should be calculated based on the address of the function, already called from `libc` at the time of exploitation, i.e., which address is already recorded in `.plt.got`.

To protect `.plt.got` from being overwritten, there is `LD_BIND_NOW` flag, which disables lazy binding and tells the loader to bind all imported functions immediately [63]. However, reading from `.plt.got` is still possible and one can calculate the address of `system` based on the address of any imported function.

Kirsch et al. [64] showed that even with turned on protections, the dynamic loader (POSIX) leaves in the program writable pointers to the functions that are called when exiting the program. The attacker can overwrite these pointers so that malicious code is executed during the program exit.

3.3. Using Gadgets from a Randomized Library

The gadgets in the vulnerable executable file are not always sufficient to implement the malicious payload. For example, there may be no gadgets to load function arguments passed through registers. Ward et al. [65] proposed a method that allows using gadgets from dynamically linked libraries, whose base addresses are randomized. It is assumed that the base address of the vulnerable executable file is not therewith randomized.

The idea is based on the ability to partially overwrite pointers from the global offset table 3 (GOT [54]). Such overwriting can be performed, for example, if a write-what-where [50] condition is present. The table 4 contains the values of pointers to code in the process memory (usually located in libraries). By changing the last byte of the pointer value one can address the code within the memory paragraph around this pointer. For example, if the pointer value is 0xdeadbeef, then an address range 0xdeadbe00-0xdeadbeff is available for addressing. Address space randomization, operating on the level of changing virtual memory tables, only changes the high-order bytes of addresses. Thus, the code located on one page does not change its low-order bytes. It follows that rewriting the low-order byte of the pointer to the code allows positionally independent addressing of the code within a memory paragraph of size $2^8 = 256$ bytes.

After the low bytes of the pointers in the Global Offset Table (GOT) are corrected, control should be transferred to them. To achieve this, one fills the stack with pointers to the records of the procedure linkage table 5 (PLT [54]), which indirectly transfers control to the addresses recorded in the corresponding cells of the GOT table. It is worth noticing that one can only use records of those functions whose addresses were filled by a dynamic loader (i.e., those called at least once before exploitation). One can call gadgets that lie within the same memory paragraph from the beginning of the function. Thus, one can call gadgets from a randomized library.

3.4. Stack Pivot

Dino Dai Zove [66] introduced a *trampoline gadget* (*Stack Pivot*), which can be used when exploiting stack or heap buffer overflows as an intermediate link. The stack pivot moves the stack pointer to the beginning of the ROP chain and thereby transfers control to it. Stack Pivots are as follows:

- `mov esp, eax ; ret`
- `xchg eax, esp ; ret`
- `add esp, <constant> ; ret`
- `add esp, eax ; ret`

The attacker can replace the function pointer with the address of the stack pivot, for instance, using a fake virtual functions table 6 formed on heap. Instead of calling the function, the stack pivot moves the stack pointer to the beginning of the ROP chain.

3.5. Canary Bypass

The compiler uses canaries [67] to protect against stack buffer overflow exploitation. During the function call, the compiler inserts an arbitrary value just before the return address on stack. This value is called a “canary”. The compiler prepends the return from the function by a code that checks the canary value. If the value has changed, the program crashes. Thus, it becomes impossible to place the ROP chain starting from the return address and execute it because that overwrites and changes the canary value.

Fedotov et al. [53] showed how to bypass the canary when the data execution prevention is operating. The method can be applied if there is a write-what-where condition [50]:

1. Buffer overflow causes overwriting of the pointer placed on stack.

2. The attacker controls the value that is written by this pointer.

Suppose that after overflow and before checking the canary, `free` function is called (Fig. 3). Then, the attacker overwrites the pointer with the address of the GOT table cell in which the address of `free` function is stored. To the cell of `free` function in the GOT table one writes the address of the stack pivot which moves the stack pointer and transfers control to the ROP chain located up the stack, but before the canary. Thus, instead of calling `free` function, control is transferred to the stack pivot that, in turn, transfers control to the ROP chain. As this occurs, the canary does not change, and verification of its value upon return from the function is successful.

3.6. Disabling DEP and Transferring Control to a Regular Shellcode

A two-stage method for exploitation is common [66] with ROP as the first stage. It is responsible for placing the second stage shellcode, disabling protections, and transferring control to the injected shellcode. The second stage executes a regular shellcode which contains the main malicious payload. Thus, it is possible to modify the malicious payload by replacing just the second stage shellcode. The following is a detailed description of both stages:

1. **ROP stage.** An attacker places a shellcode on stack or writes it into memory with a ROP chain. Next, the attacker makes up a ROP chain that disables DEP: calling the `mprotect` function [68] (`VirtualProtect` [69]) makes the injected shellcode executable. As a result, control is transferred to a regular shellcode.

2. **Shellcode stage.** The malicious payload is contained in the shellcode, which is now executable. Execution of the shellcode completes the exploitation.

Peter Van Eeckhoutte [70] described a way to bypass DEP in 32-bit Windows programs with gadget `pushad ; ret`. Registers are pre-initialized with values so that a regular ROP chain is on stack after executing the `pushad` instruction (which saves general-purpose registers onto the stack). In turn, the ROP chain calls the `VirtualProtect` function to make the stack executable, and transfer control to the regular shellcode located up the stack. A detailed description of this method and the Figure can be found in [60].

4. BYPASSING DEP AND ASLR WITH ROP

Under certain conditions, it is possible to build a code reuse attack on an application whose binary code is missing from the attacker. Bittau et al. [71] give an example of such attack – BROP (blind return-oriented programming). In this paper, the attack model assumes that the attacked web service handles each incoming request in a separate process, which is generated by `fork` system call. It means that the memory layout of the processes handlers does not change. This fact allows the attacker to learn the attacked web service dynamically.

The authors of this work show that under such conditions it is possible to dynamically search for gadgets in the memory of the attacked process. The gadget search is carried out by observing the side effects of the attacked program execution. Instead of the return address, the test value is placed on stack of the function containing the buffer overflow. Upon exiting the vulnerable function, control is transferred to this address. Conceptually at this moment, two fundamental events can occur: a crash or a pause. A crash happens when any address that is beyond the executable memory areas is given. A pause occurs due to a delay in program execution, for instance, after calling `sleep` function. Both events are easily observed through the connection status with the server. The connection closes or remains open for a while, respectively. The pause instruction address is essential for the described attack method since it allows one to find and classify ROP gadgets dynamically.

The attacker finds:

1. S – the address pausing program execution.

2. T – the address knowingly causing the program crash.

For gadget search one takes the trail address P. If the chain composed by the attacker with the trial address leads to a pause with a further crash, then the attacker assigns the trial gadget to a certain class. Below are examples of such chains:

- P, S, T, T ... – finds gadgets that do not load the values from the stack such as `ret` and `xor rax, rax ; ret`;

- P, T, S, T, T ... – finds gadgets that load only one word from the stack such as `pop  rax ; ret` and `pop rdi ; ret`;

A specific determination of the registers used by each load gadget is performed according to the side effects of function calls from the linkage table (also detected by a special procedure [71]) and system calls (`syscall`). The ultimate goal of building a chain is to call the `write` system call, which reads the image of the executable file from memory and sends it to the attacker via network. The attacker analyses the executable file in detail and constructs the ROP chain that performs the necessary actions.

Snow et al. [72] proposed another example of building a ROP chain for an application whose binary code is unknown to the attacker – JIT-ROP. A distinctive feature of this work is the conditions of a model attack. The authors believe that the attacked system contains a set of modern protection tools, such as DEP, ASLR, and even fine-grained address space randomization at each run of the application [73]. However, the authors also believe that in the application under attack there are many leaks that reveal the address space of the process. Such attack example is described for IE browsers. The attack is implemented, for example, through malicious JavaScript code loaded by the browser together with the web page. Then the attacker can build the ROP chain just during the attack. All search and classification methods should be lightweight enough to place them in the script code so that they do not load the attacked computer critically. With this purpose in mind, the authors adapted the algorithms proposed by Schwartz et al. [6], replacing the classification method with a heuristic algorithm that works reasonably well in the presence of a huge amount of binary code in the browser address space.

Göktas et al. [74] proposed an approach to the ROP chain forming that works in the presence of DEP and randomization of the binary image and all libraries. They called the basic idea of their approach the "massage" of the stack. The key point is that code pointers are written to the stack during execution (at least return addresses, and sometimes local variables containing pointers to functions). The recorded data is not cleared during the return from the function and remains there until further calls that simply overwrite them with some new values. Moreover, uninitialized local variables leave values written on stack by previous calls. As a result, with carefully selected input data, the attacker forms in space, located below the stack of a vulnerable function, a sequence of pointers to the code, interspersed with a place for data. Using the vulnerability of writing a single value outside the array one corrects the low bytes of pointers so that they point to ROP gadgets. If necessary, gadget parameters are subject to the same changes. When building chains, the authors, by analogy with the work of Ward et al. [65], are limited to a memory paragraph with respect to pointers located on program stack. This forces them also to use gadgets ending in `call` instructions. The main disadvantage of this method is the complexity and non-automation of gadget search procedure and creation of such a program execution path, which would set the values lower on stack so that a skeleton of the future ROP chain would be formed there.

5. JUMP-ORIENTED PROGRAMMING

Jump-oriented programming [17] (JOP) chains gadgets ending with instructions for jumping to the address controlled by the attacker. In x86 case, it is such instructions as `jmp eax` and `jmp [eax]`. Jump-oriented programming differs by more complicated procedure of transferring control from one gadget to another compared to return-oriented programming.

Bletsch et al. [17] arrange the control transferring for JOP as follows:

- A special gadget type, called a *dispatcher gadget*, links functional gadgets. This gadget manages a virtual program counter and executes a JOP program by moving this counter from one functional gadget to another. The functional gadgets addresses are stored in the dispatch table. Some fixed register is used as a program counter, whose value indicates the cell in dispatch table with the current gadget address. For example, such gadget is `add edx, 4 ; jmp [edx]` (`edx` – virtual program counter).
- Gadgets that perform primitive computing operations are called *functional*. Each functional gadget must return control to the dispatcher gadget. For example, control return can be implemented by jump through the register whose value is always equal to the dispatcher gadget address (`pop eax ; jmp esi` – is a gadget that loads `eax`).

Figure 4 schematically shows a control transfer model in the jump-oriented chain 4b compared with the return-oriented chain 4a on the example of a payload that invokes `exit(0)` system call. A JOP chain consists of a set of gadget addresses and their parameter values stored in memory. It is a dispatch table with the described control flow. Gadget addresses are essentially opcodes in a virtual machine defined by the memory state of the attacked process. The ROP chain is stored on stack, and `esp` register acts as a program counter. JOP differs because the chain location and the register acting as program counter can be anything. In the case of JOP in Fig. 4b, the dispatcher gadget is an instruction sequence `add edi, 8 ; jmp [edi]`. The `edi` register acts as a program counter, which increases by 8 each time the dispatcher is invoked. The JOP chain is stored in memory. It is a dispatch table, with the aid of which the dispatcher gadget sequentially invokes functional gadgets (G1, G2, G3, G4). Each functional gadget ends in `jmp ESI`. It allows one to initially set the value of the `ESI` register to point to the dispatcher gadget.

Formally, the dispatcher gadget can be described as follows:

$$pc \leftarrow f(pc); goto * pc;$$

Where a pc can be a register or some address in memory, and $f(pc)$ can be any function that changes pc in a predictable and monotonous way.

Works [9, 18] used the JOP trampoline gadget to transfer control from one functional gadget to another (do not confuse with Section 4). The following instructions can act as a trampoline gadget: `pop eax ; jmp [eax]`. This gadget alternately calls functional gadgets. Functional gadgets can be the following instruction sequences:

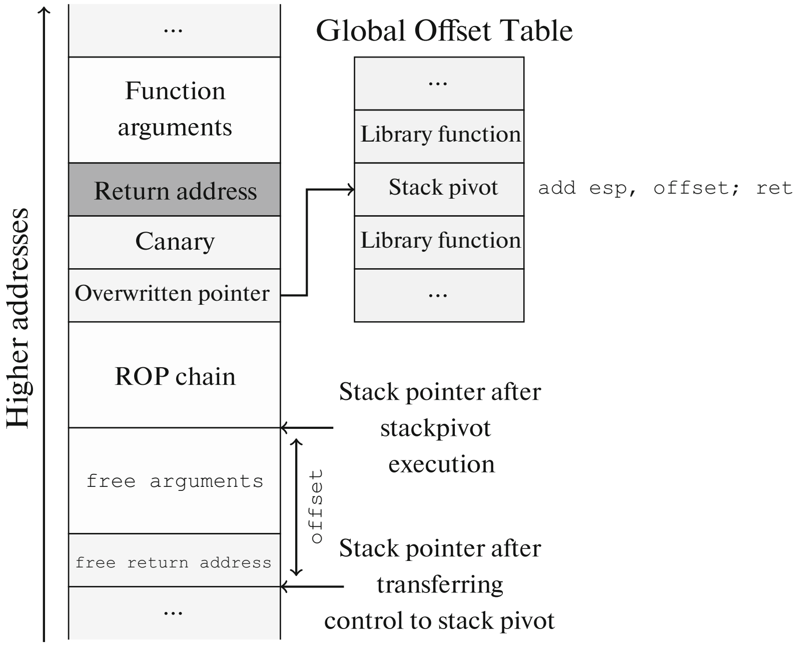


Fig. 3. Stack canary bypass.

```
pop ebx ; jmp [edx]
pop ecx ; jmp [edx]
add ebx, 4 ; jmp [edx]
```

Each functional gadget performs some basic computing operation and is sure to return control to the trampoline gadget. To do this, in the given example of instructions, the address of memory is stored in `edx` register, where the address of the trampoline gadget is stored. With this approach, the register pressure increases, which may complicate the construction of chains for x86. However, it may not be a problem for some other architectures with a large number of registers.

On ARM, the following gadget may act as a trampoline: `adds r6, #4 ; ldr r5, [r6, #124] ; blx r5`.

Chen et al. [18] showed that from a set of JOP gadgets from real libraries, a Turing-complete set of instructions can be built.

Apart from jump instructions, at least on the x86 architecture, there are call instructions `call eax` and `call [eax]`. Gadgets that end with such instructions can also be used at the end of ROP and JOP chains [17]. However, the problem of building a chain of only such gadgets needs a special approach described in PCOP [19]. Sadeghi et al. [19] show that the direct application of the method for chain building with a dispatcher gadget, as in Bletsch et al. [17], is not applicable in this case. When executing a call instruction, two operations take place:

1. The address of the next instruction (return address) is pushed onto the stack.

2. Control is transferred to the address specified in the instruction.

Using gadgets that end with a call, the main problem is the return addresses placed onto the stack. They should be removed from the stack by subsequent gadgets. Sadeghi et al. [19] proposed using *strong trampoline gadgets* for this. Such gadgets should be located between functional gadgets to transfer control and clear the stack of useless return address values. In a PCOP chain of *n* functional gadgets, it is necessary to place $n - 1$ strong trampoline gadgets between functional gadgets. An example of a strong trampoline gadget can be `pop x ; pop y ; call y`. Naturally, Sadeghi et al. [19] demonstrate the Turing completeness of a set of such gadgets from the `libc` library.

6. SIGRETURN-ORIENTED PROGRAMMING

Bosman et al. [23] proposed a method for signal handling exploitation in Unix family operating systems. When a signal is delivered to a process, the kernel saves its context (registers, stack pointer, processor flags, etc.) in the signal frame (in user space). As the return address of the signal handler, the kernel places a pointer to the code that executes the `sigreturn` system call, which restores the process context from the signal frame. An attacker can form a signal frame and call `sigreturn` to change the context of the process. Thus, using only one gadget that performs the `sigreturn` system call can write arbitrary values to the registers. This approach is called sigreturn-oriented programming (SROP) and allows Turing-complete computations. The authors offer two types of attack:

1. **The `execve` system call.** The attacker places `execve` string arguments on stack and forms a signal frame with pointers to these strings. Thus, the `sigreturn` gadget initializes the registers with arguments of the system call. Moreover, the `syscall` gadget, which is already contained in `sigreturn` gadget code, invokes the `execve` system call.

2. **Fast** `vsyscall`. Operating systems with a Linux kernel version up to 3.3 use the fast system call mechanism known as `vsyscall`. The `vsyscall` implementation code contains a set of useful gadgets at fixed addresses. In particular, a `syscall` gadget. The authors claim that gadgets remain at the same addresses after kernel security updates and on different distributions. Moreover, the same memory page as the `vsyscall` code, keeps the current time, whose lower bits, with due patience, can be used as a gadget.

7. CONTROL FLOW INTEGRITY

One way of counteracting code reuse attacks is to provide control-flow integrity (CFI) checking. There is a wide range of publications on this subject that offer one way or another to implement this method [75]. Some of the proposed implementations are even included in the compilers as test extensions, but they are not used by default for performance reasons. The general idea of these methods is that during a code reuse attack, the control flow usually differs significantly from the regular execution control flow. To detect such deviations, one can build some control flow behavior model and check its compliance with actual control transfers during execution. Theoretically, the control flow integrity prevents the application from being exploited by the code reuse methods described in previous chapters.

8. USING GADGETS THAT IMMEDIATELY FOLLOW CALL INSTRUCTIONS

In standard programs, after return from a function, in most cases, control is transferred to the instruction that goes right after this function call instruction. In general case, return-oriented programming violates this condition, transferring control to arbitrary places of the program after the return instruction. To counteract exploitation by ROP chains some CFI implementations check that return instruction transfers control to instruction right after the function call instruction. However, Carlini et al. [76] noted that in this case, one could use only those gadgets that begin right after the call instruction (CPROP, Call-Preceded ROP). These gadgets appeared to be larger, having more complicated side effects. However, the authors showed that real applications contain these gadgets in quantity sufficient to create workable ROP chains with a careful account of their side effects. The control flow when using CPROP gadgets is shown in Fig. 5a by a solid line, and the dashed line shows the original control flow.

9. REUSE OF FUNCTIONS

In order to bypass CFI, specific code reuse attacks are proposed. For example, in simplest cases, complete functions are enough as gadgets. Tran et al. [77] studied the question: how expressive the return-to-library attacks are. They show that return-to-library attacks are, in fact, Turing-complete. They build a Turing-complete set of *widgets* from POSIX compatible functions of the C standard library to prove this. A *widget* is a function with beneficial side effects, so it is an analog of a gadget. To implement branching, one uses `longjmp()` widgets that change the stack pointer. Based on a set of widgets, two exploit examples are built. They show the practical applicability of the proposed approach. Moreover, since this set of widgets is built of POSIX-compliant functions, widget chains are portable between POSIX-compliant operating systems. Table 2 demonstrates an example of POSIX-compliant widgets.

It is more challenging to build widget chains than ROP chains manually due to complex data and control dependencies. Besides, the widget chain requires a larger stack due to the size of the chain itself.

It is worth noticing that building of exploits consisting of widgets alone is possible on the x86 architecture only with calling convention, in which the function arguments are passed through the stack. Otherwise, one should use ROP gadgets to load the arguments.

Lan et al. [78] propose a loop-oriented programming (LOP) method that uses whole functions as gadgets. This method was developed to bypass coarse-grained CFI and the shadow stack [79]. In order to bypass such security features, it is necessary to transfer control to the beginning of the function and return control to the calling function at the point immediately after the call site. The method vaguely resembles jump-oriented programming (Fig. 4b). The authors take whole functions (functional gadgets) as gadgets,

Table 2. A set of POSIX-compliant widgets

Category	Widgets

Branching	`lfind() + longjmp(),`
	`lsearch() + longjmp()`

Arithmetic/Logic	`wordexp(), sigandset(),`
	`sigorset()`

Memory access	`memcpy(), strcpy(),`
	`sprintf(), sscanf(), etc.`

System calls	`open(), close(), read(),`
	`write(), etc.`

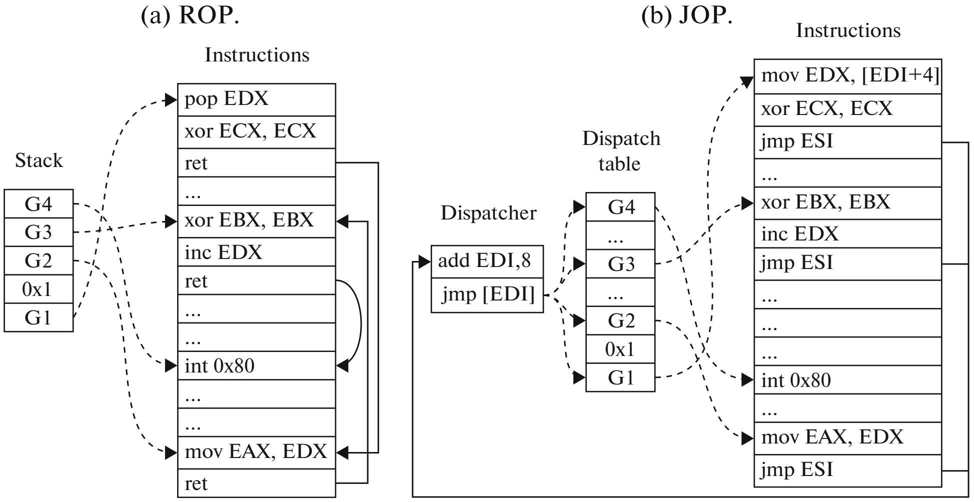


Fig. 4. Comparison of control flow for ROP and JOP with example chain invoking exit system call.

while the dispatcher gadget transfers control between them. A function with a cycle acts as a dispatcher gadget. Inside the loop, there is an indirect call. The functional gadgets addresses are stored in the dispatch table. At each iteration of the loop, the dispatcher gadget monotonously changes the virtual program counter and calls the next gadget by the address from the dispatch table, pointed by the counter. After return from the functional gadget, control is transferred to the next iteration of the dispatcher gadget. Figure 4b shows how control is being transferred between functional gadgets through the dispatcher gadget.

In object-oriented languages, primitives similar to dispatcher functions can often be met, for example, iterating over collections of same type objects, with a specific virtual method being called for each of them. For such cases, COOP [80], LOOP [81] proposed a method for exploitation that replaces the virtual call table. Properly created virtual tables of objects from the collection allow one to organize a chain of objects methods calls. In this case, by analogy with the previous approaches of the current section, whole functions act as gadgets. Data can be transferred between different gadgets either via common fields of objects or through uninitialized variables. In procedural languages, one may also encounter similar iterations over collections of structures that contain pointers to processing functions. For an application written in C as an example, FOP [82] shows an example of building an exploit from program functions.

10. DATA FLOW EXPLOITATION

Chen et al. [83] were the first to show that it is possible to exploit the application data flow and to execute a payload in its context without breaking the control flow integrity. The authors presented several real and synthetic examples. Later, Hu et al. [84, 85] gave the name to these types of attacks (DOP, Data-Oriented Programming). They also showed that DOP attacks could be Turing-complete, i.e., they can execute arbitrary code without violation of the program control flow integrity. Thus, such attacks can not be detected by control-flow integrity methods.

When building DOP chains, one uses DOP gadgets that can be arbitrary pieces of code, and a special dispatcher gadget which is necessary to transfer control between DOP gadgets. The instructions of the virtual machine in which the DOP chaining is executed are some instruction sequences in the original program. The values of the DOP chain variables are stored in memory, since the registers are actively used in the program and tend to get corrupted between executions of two consecutive DOP gadgets. An example of a dispatcher gadget is a cycle with some mechanism to select a DOP gadget, which allows the current gadget to transfer control to the next gadget.

Hu et al. [84, 85] semi-automatically created DOP chains. One should find all DOP gadgets and a dispatcher gadget to build a DOP chain. Having discovered them, one should find the input data for the target program, which leads the execution path to the place containing the gadgets found. Furthermore, each gadget is supplied with information about which region of memory it has changed (global variables, function parameters, local variables). The easiest way is to use gadgets that change the state of global memory. When there are memory errors in the program, attack designing is divided into the following stages: searching for gadgets, selecting suitable gadgets, and stitching them. To stitch gadgets, Hu et al. [84] build a 2D data flow graph that represents data flows in two dimensions: memory addresses and runtime. Then they try to discover new edges on this graph. The authors show some examples of designed attacks on real applications that bypass DEP, ASLR, and CFI protections. Besides, they show that some real applications contain enough gadgets to create DOP chains, including Turing-complete ones.

In works [86, 87], DOP and methods for the automated generation of DOP chains got further development. Despite the similarity of the main idea of this type of attacks, the methods for the automated generation of such chains are significantly different. Their detailed discussion is beyond the scope of this article.

11. GENERAL CODE REUSE EXPLOIT GENERATION SCHEME

We schematically divide the process of code reuse exploits generation into four stages:

1. The gadget search in non-randomized executable areas of the process memory image (Section 13).

2. The gadget semantics determination (some methods may skip this stage). At this stage, the payload that each gadget performs is determined (Section 14).

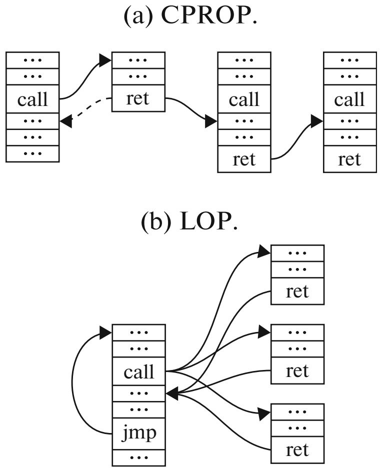
3. The combination of gadgets and their parameters to obtain a chain of gadgets that performs a given sequence of actions (Section 15).

4. The automated exploit generation [53, 88–91] – generation of input data causing the program exploitation by injecting and executing the ROP chain. At this stage, machine instructions (on the program execution trace from the point of receiving input data to the vulnerable point) are symbolically executed [92–94]. Thus, a path predicate is constructed. The path predicate is united with the security predicate that describes the ROP chain injection and transferring the control to it. The solution to the obtained system of equations is an exploit. The path predicate ensures that the program runs the same path to vulnerability, and the security predicate provides control flow hijacking.

12. GADGET CATALOG

Before discussing specific methods for searching and determining the gadgets semantics, let us define the *gadget catalog* as a list of entries with the following contents:

1. **Semantic description** of the instruction sequence. Each description usually corresponds to some basic computational or memory operation (addition, subtraction, writing to memory, reading from memory, initializing the register with some value, transferring control, etc.).


(a) CPROP.

(b) LOP.

Fig. 5. The control flow of CPROP and LOP. All blocks represent functions.

2. **Virtual address** of the gadget found in the application address space. It is an operation code for the instruction set architecture that is defined by the gadget catalog.

3. **Machine instructions** of the gadget are a specific instruction sequence that implements a given semantic description. They can be cataloged manually or automatically during application binary image analysis.

4. **Gadget parameters** are parameters of the semantic description (specific registers, constants, etc.).

5. **Side effects** of gadget execution relatively to its semantics. A side effect is any change in memory and registers, not described by the gadget semantics. Side effects can be cataloged manually or automatically calculated during gadgets classification.

We give an example to clarify this definition. Table 3 presents the gadgets catalog, consisting of several semantic descriptions. The first semantic description corresponds to the operation of adding the values of two registers $r1 + = r2$. The second semantic description corresponds to the instruction for loading a value from the stack into the register. The last semantic description defines the gadget loading three registers

Table 3. Incomplete gadget catalog.

Semantic description	Virtual address	Machine description	Gadget parameters	Side effects
r1 += r2				
r = M[ESP + Offset]				

r1 = M[ESP + Off1]
r2 = M[ESP + Off2]
r3 = M[ESP + Off3]				

Table 4. Complete gadget catalog.

Semantic description	Virtual address	Machine description	Gadget parameters	Side effects

r1 += r2	0xdeadbeef	add eax, ebx
		pop edx
		ret
			r1 = eax
			r2 = ebx
				edx ✗

	0xcafecafe	add eax, ebx
		pop ecx
		ret
			r1 = eax
			r2 = ebx
				edx ✗

	0xcafebabe	add edx, ecx
		ret
			r1 = edx
			r2 = ecx
				–

r = M[ESP + Offset]	0x12345678	pop eax
		ret
			r = eax
			Offset = 0
				–

r1 = M[ESP + Off1]
r2 = M[ESP + Off2]
r3 = M[ESP + Off3]
	0x10203040	pop eax
		pop ebx
		pop ecx
		ret
			r1 = eax, Off1 = 0
			r2 = ebx, Off2 = 4
			r3 = ecx, Off3 = 8
				–

from the stack. After the gadget search, the catalog takes the form shown in Table 4. The first two gadgets, found at addresses 0xdeadbeef and 0xcafecafe, have side effects relative to the basic semantic description because they change values of `edx` and `ecx` registers respectively.

12.1. Turing-Complete Gadget Catalog

The authors of several works [5, 7, 18, 19, 77, 95] compose a gadget catalog in such a way that the set of semantic descriptions is Turing-complete. Such gadget catalog defines some new computing machine capable of performing arbitrary computations.

While searching and determining gadgets semantics, their addresses are being cataloged. After that, two situations are possible:

1. For each semantic description, a specific gadget has been found.

2. There are no specific gadgets for some semantic descriptions.

In the first case, it turns out that the found set of addresses implements the computer described by the catalog on a specific executable file. It means that it is possible to make arbitrary calculations with gadgets found. Moreover, one can use this catalog to describe the target architecture instruction set for the C compiler (llvm [7]).

In the second case, when there are no gadgets for some semantic descriptions in the Turing-complete gadget catalog, arbitrary calculations can no longer be performed. Therefore, the question arises: what is the computational power of found gadgets set? In other words, is it possible to execute a given program with the found set of gadgets? To answer this question, each program that describes the exploit should be analyzed separately. Based on the operations that it performs and the gadget catalog contents, in some cases, it can be concluded before the exploit generation that generation is impossible. For example, there are conditional transitions in the initial exploit program, while there are no gadgets that implement branching in the gadget catalog. A similar situation is with writing to memory. In other cases, one needs to try to build an exploit from existing gadgets. If it is built, then the answer to the question is positive and is confirmed by the designed exploit. Otherwise, the generation task can be reduced to an exhaustive search of all possible combinations of gadgets, which can be time-consuming for a large gadget catalog. We suppose that the infinite exhaustive search with no possibility of generating an exploit is rare for real executable files.

In order to compile Turing-complete ROP chains, one should be able to conditionally change the stack pointer that acts as a program counter. Roemer et al. [7] propose the following way to implement conditional branching for the x86 architecture:

1. Perform some operation that updates the interesting flag.

2. Copy the interested flag from the flag register to the general-purpose register.

3. Use this flag for conditional changing the stack pointer by the desired offset (for instance, by multiplying the offset by the flag value 0 or 1).

13. GADGET SEARCH

Regardless of the exploit building process, one should first find all available gadgets in the binary image of the application. There are two fundamental approaches to the gadget search task. The first of them offers to search for gadgets by the list of templates. Templates are usually specified by regular expressions over binary codes of gadget commands. Initially, the gadget catalog contains semantic descriptions of gadgets. For each semantic description, gadgets are searched according to some template. As a result, specific gadgets are included in the gadget catalog for corresponding semantic descriptions: virtual addresses, machine instructions, and gadget parameters. Side effects (for instance, clobbered registers [34, 38]) can be obtained upon analyzing the machine instructions of found gadgets.

The second approach is automatic search for all possible instruction sequences ending in a control transfer instruction. The classic algorithm that implements the search for all gadgets is Galileo algorithm [5]. It first looks for control transfer instructions in the executable sections of the program. For each instruction found, it tries to disassemble several bytes preceding the instruction. All correctly disassembled instruction sequences are cataloged. Thus, the catalog contains virtual addresses and machine instructions for gadgets. Many open-source gadget search tools use this algorithm [32–38, 44, 96–99].

14. DETERMINING GADGET SEMANTICS

Not all found gadgets are suitable for building ROP chains. One should understand what useful payload this gadget performs in order to use the gadget in building the ROP chain. The gadget semantics can be determined manually [5]. In the template search for gadgets, the semantics are contained in the template description [7, 9, 13, 17–19, 25, 74, 95].

14.1. Gadget Types

Schwartz et al. [6] proposed defining the functionality of the gadget by its belonging to some parameterized types that define the new instruction set architecture (ISA). Type parameters are registers, constants, and binary operations. In order to use the gadget when building ROP chains, one should fulfil the following gadget properties:

- **Functional.** Each gadget has a type that determines its functionality. The type of gadget is described semantically with the postcondition – the Boolean predicate $\mathcal{B}$. It must always be true after the gadget is executed. It is worth noticing that one gadget can belong to several types simultaneously. For example, the gadget `push eax ; pop ebx ; pop ecx ; ret` simultaneously moves `eax` into `ebx` and loads the value from the stack into `ecx`, which corresponds to `MoveRegG: ebx` ← `eax` and `LoadConstG: ecx` ← `[esp + 0]`.
- **Control preserving.** Each gadget should be able to transfer control to another gadget.
- **Known side effects.** The gadget should not have unknown side effects. Side effects of gadget execution should not lead to uncontrolled program behavior. For example, writing a value to an arbitrary memory address may cause a program crash.
- **Constant stack offset.** Most gadget types require the stack pointer to increase by a constant value after each execution.

In order to determine whether the gadget instruction sequence $\mathcal{I}$ satisfies the postcondition $\mathcal{B}$, Schwartz et al. [6] use a well-known technique from formal verification, namely, computing the weakest precondition [100]. At a high level, the weakest precondition $wp(\mathcal{I}, \mathcal{B})$ for an instruction sequence $\mathcal{I}$ and condition $\mathcal{B}$ is a boolean precondition that describes when $\mathcal{I}$ is completed in the state satisfying $\mathcal{B}$.

The weakest precondition is used to ensure that the gadget semantics are always determined after the execution of the instruction sequence $\mathcal{I}$. For this purpose, it is sufficient to check:

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

If the formula is correct, then $\mathcal{B}$ is always true after performing $\mathcal{I}$, which means that $\mathcal{I}$ is a gadget with semantic type $\mathcal{B}$.

However, formal verification of the gadget semantics practically appeared to be very slow. The authors proposed a combined approach to speedup the process. The gadget instructions are preliminarily executed several times on random input data, and then the truth of $\mathcal{B}$ is verified. If B turns out to be false for at least one execution, then the instruction sequence cannot be a gadget of this type. Thus, a more complicated computation of the weakest precondition is performed only if $\mathcal{B}$ is true for each execution.

The combined approach can be conventionally divided into two stages: gadget classification and gadget verification. At the classification stage, hypotheses are made that gadgets belong to certain types and about the values of these types parameters. Hypotheses are essentially defined by Boolean postconditions. And at the verification stage, for each postcondition, its truth or falsehood is formally proved, and the hypothesis is accepted or rejected, respectively.

14.1.1. Gadget Classification. Currently, there are a lot of processor architectures with various instructions. In order to abstract from the specifics of a particular architecture when writing universal algorithms, one traditionally uses an intermediate representation of machine instructions (VEX [101], REIL [102], Pivot [103, 104], etc.). In this case, the binary code analysis algorithms work with a simpler intermediate representation, and not with the target processor architecture.

Table 5. Verification of gadget `ArithmeticLoadG: ebx ← ebx + [eax]`.

Step	Symbolic state	Instruction	Set of formulas
initial	M, $eax = \phi_1$, $ebx = \phi_2$, $ecx = \phi_3$, $esp = \phi_4$, $eip = \phi_5$	–	$S_0 = \varnothing$
1	$ecx = \phi_6$	`mov ecx, [eax]`	$S_1 = S_0 \cup \{\phi_6 = (concat$ $(select\ M\ \phi_1)$ $(select\ M\ \phi_1 + 1)$ $(select\ M\ \phi_1 + 2)$ $(select\ M\ \phi_1 + 3))\}$
2	$ebx = \phi_7$	`add ebx, ecx`	$S_2 = S_1 \cup \{\phi_7 = \phi_2 + \phi_6\}$
final	$eip = \phi_8$, $esp = \phi_9$	`ret`	$S_3 = S_2 \cup \{\phi_8 = (concat$ $(select\ M\ \phi_4)$, $(select\ M\ \phi_4 + 1)$, $(select\ M\ \phi_4 + 2)$, $(select\ M\ \phi_4 + 3))$, $\phi_9 = \phi_4 + 4\}$
Determining semantics			**Verification**
verify	$final(ebx) = initial(ebx) + initial(M[eax])$		$\phi_7 \neq \phi_2 + (concat$ $(select\ M\ \phi_1)$ $(select\ M\ \phi_1 + 1)$ $(select\ M\ \phi_1 + 2)$ $(select\ M\ \phi_1 + 3))$ is **UNSAT**

In papers [60, 105], gadget classification is based on the interpretation of gadget instructions intermediate representation. During interpretation, accesses to registers and memory are tracked. If the first reading of a register or memory area occurs, the value read is randomly generated. As a result of the interpretation, the initial and final values of the registers and memory are obtained. Based on this information, possible gadget types are guessed. For example, for membership in the `MoveRegG` type [6], such pair of registers should exist, that the initial value of the first register equals to the final value of the second one. The analysis results in a list of all types satisfying the gadget, and their parameters (list of candidates). Then, several more runs of the interpretation process with diverse input data are performed. Thus, erroneously determined types are removed from the list of candidates.

Moreover, the gadget classification may result in the following [60]:

- The list of "clobbered" registers, whose values changed during the gadget execution.

Listing 1 Incorrectly classified gadget that is rejected after verification.

```
neg eax ; sbb eax, eax ; and eax, ecx ;
pop ebp ; ret
MoveRegG: EAX   ECX
```

- Information about a gadget frame (Section 1): the frame size and the offset of the cell with the address of the next gadget relative to the beginning of the frame.

It is worth noticing that the number of incorrectly classified gadgets can be reduced by adding runs of the interpretation process with boundary input data 0 and −1. The percent of incorrectly classified gadgets, in this case, is insignificant and amounts to 0.7% [106].

14.1.2. Gadget Verification. Gadget classification is a set of postconditions that describes the possible gadget semantics. Gadget verification allows one to formally prove the truth of these postconditions for arbitrary input data. The gadget can be verified by the weakest precondition [6, 28] or through a symbolic execution of the gadget instructions [105–107].

We consider the method for gadget verification through symbolic execution [92–94] in detail. During symbolic execution, the gadget semantics are modelled with SMT [108] expressions. Initially, all registers are assigned to free symbolic variables. The symbolic memory at the beginning is an empty byte array M of bit vectors:

$$M = (Array(\_\ BitVec\langle addrSize\rangle)(\_\ BitVec8)),$$

where $\langle addrSize\rangle$ – the dimension of the architecture address word. The symbolic state contains a mapping from registers to symbolic variables and the current state of symbolic memory. Symbolic execution of gadget instructions generates SMT formulas over variables and constants, and also updates the symbolic state of registers and memory in accordance with the instruction operational semantics. Operation with symbolic memory is performed through *select* and *store* operations on *Array*. The function (*select M i*) returns the i-th element of the array M and simulates reading a byte at address i. The function (*store M i b*) returns the array obtained from the array M by storing element b at index i, which simulates a record of byte b at address i.

Heitman et al. [105] first translate the gadget instructions into an intermediate REIL representation [102]. And only after that, REIL instructions are subjected to symbolic execution.

Verification postcondition is a Boolean predicate over initial and final values of registers and memory. Registers and memory from the corresponding symbolic states are substituted in the predicate. The validity of the postcondition formula is verified through the unsatisfiability of its negation with the SMT solver.

Table 5 shows an example of `ArithmeticLoadG: ebx` ← `ebx + [eax]` verification. Initially, registers are assigned to free symbolic variables ϕ_i, and an array represents the memory. A set of formulas is empty. New formulas are added in accordance with the operational semantics of the instruction under interpretation. Formulas are created according to SSA form – when a formula is added, a new symbolic variable is created to which this formula is assigned. At the first step, a new symbolic variable ϕ_6 is created. It is equal to the value of the second instruction operand `[eax]` loaded from memory. In the symbolic state, the `ecx` register is assigned the symbolic variable ϕ_6. At the second step, the result of addition is assigned to the variable $\phi_7 = \phi_2 + \phi_6$, which, in turn, is assigned to the resultant instruction operand – the *ebx* register in the symbolic state. At the final step, the symbolic state is updated according to the return instruction operational semantics, i.e., the instruction pointer is loaded from the stack, and the stack pointer is incremented by 4. Symbolic variables from the initial and final symbolic states are substituted in the postcondition describing the gadget type. SMT solver checks the satisfiability of the formula negation. Negation of the formula is unsatisfiable, which means the gadget satisfies the declared type with parameters.

Listing 1 shows an example of a gadget that can be misclassified, and verification fixes this error. During classification, the gadget was classified as `MoveRegG: EAX` ← `ECX`. For the nonzero initial value of the `eax` register, the gadget copies the value of the `ecx` register to `eax`. However, if the initial value of `eax` is zero, then its final value is zero, which is not a copy of `ecx` register value other than zero.

14.1.3. Gadget Cataloging. Gadgets are cataloged (Section 12) as follows. The gadget catalog initially contains semantic descriptions of gadget types. Each gadget found by the Galileo algorithm is classified. As a result of the classification, one obtains semantic descriptions (gadget types) that correspond to the gadget. Entries with gadget virtual addresses and machine instructions are appended to the corresponding semantic descriptions. Also, gadget parameters (values of type parameters) and side effects (“clobbered” registers) are cataloged in these entries. Further, all gadgets from the catalog are verified. As a result of verification, incorrectly classified gadgets are deleted from the catalog.

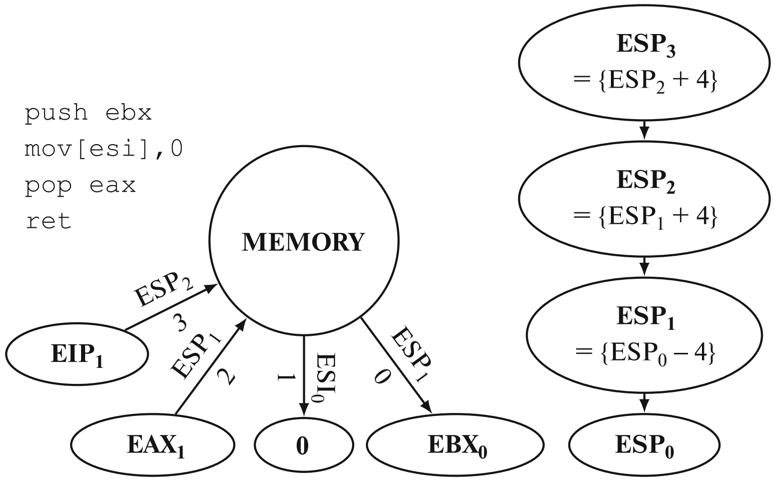
14.2. Gadget Summary

A gadget summary [8, 29, 109] is a description of gadget semantics in the form of a compact specification. Gadget summaries contain preconditions and postconditions over the values of registers and memory. In particular, the gadget summary may contain:

- registers loaded from the stack (`eax = [esp + 4]`),
- registers read from memory (`ecx = [edx + 2]`),
- registers whose values were changed (`ecx = eax + ebx`),
- ranges of memory addresses used for reading or writing (`[rsp] <-> [rsp + 0x20]`).

Follner et al. [29] proposed the following method for composing a gadget summary. First, gadget instructions rise to the level of VEX [101] intermediate representation. Then all assignments are advanced to form a single expression, called a postcondition. The postcondition describes all operations by which the final value was obtained in the register under consideration. The analysis supports a memory model that allows one to simulate the situation of passing values via the stack correctly. Also, this analysis allows one to obtain preconditions that describe ranges of memory accesses by register with an offset (`[rax] <-> [rax + 0x20]`). Preconditions indicate that registers from these memory ranges should point to memory available for reading or writing.

Gadgets cataloging (Section 12) goes as follows. The gadget catalog initially contains gadgets virtual



```
push ebx
mov[esi],0
pop eax
ret
```

Fig. 6. Gadget dependency graph.

addresses and machine instructions. A summary is composed for each gadget from the catalog, which essentially allows one to catalog gadget semantic description and side effects.

14.3. Gadget Dependency Graph

Milanov [30] proposed to represent a gadget in the form of a directed dependency graph (Fig. 6). Vertices correspond to registers, memory, and constants. All memory is represented by one node. While a register can correspond to several vertices: each modification of the register generates a new vertex (reg_0, reg_1, reg_2, etc.). Directed edges reflect data dependencies (register assignment, memory access, etc.). Gadget instructions generate new edges on the graph. The edges connected to the memory also contain tags with a memory access address and are numbered in chronological order.

Gadget cataloging (Sec. 12) goes as follows. Initially, the catalog contains virtual addresses and gadget instructions. The instructions of each gadget are translated into REIL [102] intermediate representation, for which a dependency graph is constructed. As a result of the graph traversal, a gadget semantic description is computed: the final values of registers and memory are expressed via the initial ones. An expression for some finite value may have a condition under which this expression is true. Further, the gadgets are classified by type (Section 1). The author's motivation for such method of determining the gadget semantics was a shorter working time compared to methods using SMT-solvers.

For example in Figure 6 the following semantic description is obtained:

$$EIP_1 = MEM[ESP_0]$$

$$ESP_3 = ESP_0 + 4$$

$$EAX_1 = \begin{cases} 0 & \text{if } ESP_0 - 4 = ESI_0 \\ EBX_0 & \text{if } ESP_0 - 4 \neq ESI_0 \end{cases}$$

$$MEM[ESP_0 - 4] = \begin{cases} 0 & \text{if } ESP_0 - 4 = ESI_0 \\ EBX_0 & \text{if } ESP_0 - 4 \neq ESI_0 \end{cases}$$

$$MEM[ESI_0] = 0$$

15. GADGET CHAINS GENERATION

This section describes various methods for ROP chains generation. It is worth noticing that chaining gadgets is a brute force task, therefore, to reduce the number of brute-force iterations, one can prefilter unnecessary gadgets and sort them by quality [110]. ROP chains generation differs from regular compilation in the following ways:

- Most often, a ROP chain cannot save registers values in memory for their further recovery due to the lack of relevant gadgets.
- ROP gadgets may have side effects. For example, a gadget can "clobber" registers. Values of "clobbered" registers are not saved after the gadget execution. Side effects should be considered during gadgets scheduling [6].
- Some gadget types (Section 1) that act as virtual machine instructions may not be available. In this case, it is necessary to replace the missing gadgets with a sequence of others [6].

During generation, one should consider restricted symbols that cannot be used in the ROP chain. For instance, an overflow may take place with `strcpy` function, which prevents the chain from containing zero bytes. However, only a few completely solved the problem of restricted symbols [27]. Most solutions just delete gadgets whose addresses contain restricted symbols but do not check the gadget parameter values on stack.

ROP payload can be divided into the following: setting registers to specified values and execution of one more gadget [37]. Thus, the method for ROP chains generation can be based on registers setting, and the rest of the payload can be implemented by appending to the resulting chain one gadget that writes to memory, calls a function, invokes a system call, etc.

15.1. Turing-Complete Compilation with a Fixed Gadget Catalog

Let us consider building a compiler based on a fixed gadget catalog. Buchanan et al. [7, 13] manually composed a Turing-complete gadget catalog from the machine code of Solaris OS `libc` standard library for the SPARC architecture. Each semantic description is associated with a single instruction sequence from the library machine code. SPARC allows only aligned accesses to instructions, so all gadgets are legitimate epilogues of library functions.

SPARC uses register windows. The register window consists of registers intended for input parameters, return values, and temporary values inside the procedure. When the function is called, the register window is shifted forward, and it is reversed on function return. When call stack is large, there is a lack of register windows, which makes it necessary to save them on stack. In this case, during function return, the register values are restored from the values stored on stack. This leads to an undesirable change in register values when transferring control between two gadgets. Thus, the SPARC architecture and its calling convention impose restrictions on the way the calculated values are passed between gadgets – only through memory. The gadget catalog of Buchanan et al. [13] implements a set of gadgets that use the memory-memory model, which allows the use of registers only inside gadgets, and the values are passed from one gadget to another through memory. Each variable in the ROP chain is associated with the address of the memory cell, which is used as the gadget operand.

After the gadget cataloging, there are two options to create ROP chains automatically. Firstly, the gadget catalog has a C programming interface. It contains 13 functions that allow one to create variables, assign values, and call functions (or make system calls). With this program interface, one can write a program that automatically generates a ROP chain using a gadget catalog. Secondly, Buchanan et al. [13] wrote a translator from some pseudo-language of the exploit description (narrowed C) into a sequence of function calls from the gadget catalog program interface in C language. The compiler implements most of the basic arithmetic, logical operations, operations with pointers and memory, and operations of conditional and unconditional control transfer.

Some authors [13, 28] note that it is possible to write an extension for the LLVM compiler infrastructure, which allows one to generate code for the virtual machine defined by the gadget catalog.

The tool, introduced by Mosier [31, 98], relies on ROPC-IR, an exploit description assembly-language, that defines a Turing-complete instruction set architecture. It has three registers: `ACC(eax)`, `SP(rbp)`, `PC(rsp)`, and operations for interacting with these registers: basic arithmetic (`ADD, SUB, NEG`), branch instructions (`CMP, JMP, JNE`), register-to-register (`MOV`) instructions, register-to-memory (`LD, STO`) instructions, stack instructions (`PUSH, POP, ALLLOC, LEAVE`), control transfer instructions (`CALL, SYSCALL3, LIBCALL3`). The `rsp` register acts as the program counter (`PC`). Moreover, a separate stack is allocated for the functions inside the ROP chain, the pointer to which `SP` is `rpb`.

Support for the second stack allows one to implement function calls inside the ROP chain, which implement full-fledged subprograms. Besides, it supports the ability to call functions from the target program address space and the ability to make system calls. As an illustration, Mosier [98] gives an example of ROPC-IR code for calculating Fibonacci numbers through a recursive function call from the ROP chain, the `printf` library function call, as well as the `exit` system call.

ROPC-IR language description represents the gadget catalog in this tool. The gadget search process prints all kinds of gadgets from the target program. Then it is necessary to manually find and assign specific gadget to each semantic description and manually catalog the following: virtual address, gadget parameters. By definition of the ROPC-IR language gadgets have no side effects (not taking into account output registers). Theoretically, the ROPC-IR assembler language can act as the target C compiler language. However, the practical application for building exploits for real programs can be significantly limited by the presence of the necessary gadgets in the program and the size of the generated ROP chains. Due to the nonoptimality of the ROPC-IR language translation, the chain size is significantly larger than the typical exploit sizes.

The approach to building an automated tool for generating ROP chains, proposed in [7, 13, 31], is based on a fixed gadget catalog. It is once formed by the authors and does not change. Besides, semantic descriptions are tightly bound to the specific registers used in gadgets. If some version of the standard `libc` library lacks any gadget, then ROP chain compilation may fail. In this case, it would be possible to use other gadgets that have different operands, but similar functionality, and achieve successful compilation. In other words, this approach has limited practical applicability, especially in situations of a small number of gadgets in the library code under study.

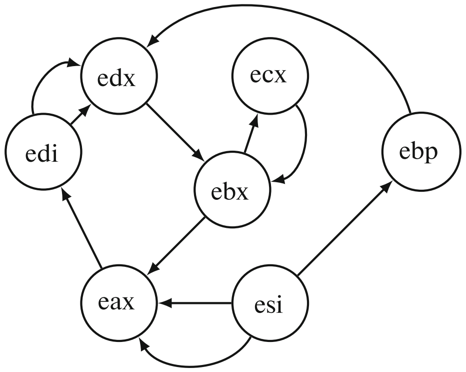


Fig. 7. MOV connection graph. Chained gadgets can be used to emulate missing gadgets.

15.2. Generation based on Gadget Templates

Generation based on gadget templates is a search by regular expressions for a specific sequence of gadgets that performs some malicious payload: the `execve` system call [35, 38], the `VirtualProtect` function call, followed by the execution of a regular shellcode on stack (Section 6) [34], etc. Such an approach can handle restricted symbols by negating the value loaded from the stack, by repeating the increment up to the desired value, or by other arithmetic operations. It is worth noticing that Ropper [?] uses SMT solvers to search for gadgets that satisfy the semantics defined by the postcondition over registers, memory, and constants. However, at the time of writing this paper, the tool uses only regular expressions to generate ROP chains.

Huang et al. [11], for the ARM architecture, use an approach based on a unique gadget which simultaneously loads the values of all registers from the stack. The searching algorithm and simultaneous checking of the gadget for compliance with the given semantics are performed by analyzing the assembler code instructions. Generating a chain from one gadget is a trivial task. It requires only the correct location of the register values on stack.

Hund et al. [25] present another approach to gadget cataloging and compilation. Firstly, they search only for gadgets consisting of one instruction, not counting the return instruction itself. Most likely, it is done in order to simplify the algorithms for analyzing the gadget parameters and side effects. Such gadgets are cataloged. Secondly, they supplement the gadget catalog with the gadgets that can be combined from existing ones. The following example can illustrate it:

1. `pop eax ; ret` – gadget loading value from stack to register `eax`,

2. `mov ebx, eax ; ret` – gadget moving the value from the register `eax` to `ebx`.

These two gadgets, called sequentially, form a gadget for loading value from the stack into the `ebx` register. Hund et al. [25] provide an algorithm to search for all possible combinations of gadgets that move a value from one register to another. This task reduces to finding a path from one vertex to another in a special graph. In that graph, registers act as vertices whereas initial gadgets, that move one register to another, act as edges. Figure 7 presents an example of such graph.

This approach allows one to expand the gadget catalog, which is especially useful for exploitable programs of small size. However, using combined gadgets, one should necessarily consider side effects at the stage of chaining gadgets.

Nguyen An Quin [26] proposed a similar idea to combine several gadgets into one that performs the desired behavior. For example, a gadget sequence `push 0x1234 ; pop ebp ; ret ; xchg ebp, eax ; ret` can be considered as one gadget loading the constant 0x1234 into the `eax` register.

15.3. Chaining Gadgets by Semantic Queries

Milanov [30] obtains a semantic description of each gadget by building its dependency graph (Section 3). Gadgets are chained together by sequential semantic queries to the gadget catalog. This method is implemented as an open-source tool ROPium [36]. A semantic query is essentially an expression over constants, final/initial values of registers and memory. First, gadgets with a semantic description that satisfy the semantic query are searched in the gadget catalog. If such gadgets are missing, then the semantic request is split into several ones according to some strategies. For example, the first register can be moved to the second through some intermediate third register.

A notable feature of the ROPium tool is support of gadgets ending in `call Reg` and `jmp Reg` instructions. A load gadget for `Reg` register is added before such gadgets. The gadget loads the gadget address to which it is necessary to transfer control after executing the gadget with `call` or `jmp`. In the case of `call`, it may also be necessary to transfer control to a special gadget that removes the return address placed by `call` from the stack.

15.4. Genetic Algorithm

Fraser et al. [12, 111] suggest a different approach to building ROP chains. The authors suggest using genetic algorithms for this. Fraser et al. introduced the ROPER tool available on GitHub [33]. The tool allows one to generate a ROP chain for the ARM architecture, which sets registers to the specified values.

Initially, gadgets are searched in the executable. For each of them, the frame size and the offset of the next gadget address are calculated (Section 1). Then, the executable file is loaded into the virtual machine address space for repeated ROP candidate chains execution. The virtual machine provides a convenient interface for executing guest architecture instructions.

During genetic mutations, gadget addresses play the role of genes. Random values are placed onto the stack as data and next gadget addresses. The fitness function is the difference between the current and the required vector of register values. Genetic mutation methods modify each element of the population. Among all the candidates, we select a set of potentially best candidates for which the mutation process is repeated.

It is worth noticing that the chains formed by the ROP genetic algorithm are very different from those created by people. For example, they can write values to their stack or transfer control to gadgets that were not originally found during the search process. Apart from that, the chain size can be large due to the suboptimal choice of gadgets. The described disadvantages may come from the lack of information about gadget instructions semantics in the genetic algorithm. Perhaps, if we consider it in some form and use more modern machine learning methods, we can develop the concept of this approach into a practically useful tool.

15.5. Chain Generation by SMT-Solvers

Follner et al. [29] proposed a generation method based on a gadget summary (Section 2), which has an available source code [32]. The method allows one to get a sequence of gadgets that writes the specified values into *m* requested registers. It is worth noticing that the method does not calculate the gadget parameters loaded from the stack, but only provides a sequence of gadget addresses. Initially, a summary is compiled for all gadgets. For each requested register, the algorithm selects *n* most suitable gadgets [110], which load the register value from the stack or memory controlled by the attacker.

Further, for various gadget chains ($n^m * m!$ combinations), preconditions and postconditions are calculated for the entire chain. If the postconditions satisfy the situation when the attacker controls values of all requested registers, then the method turns to the final stage – the preconditions resolving. Additional gadgets are prepended to the beginning of the chain. These gadgets initialize registers from the preconditions, so that they point to the memory available for reading and writing.

Algorithm 1 Search algorithm of shortest chain initializing registers

```
regset_to_chain ← empty register set mapping to shortest chains
queue ← empty queue
queue.push(empty chain)
while queue is not empty do
    chain ← queue.pop()
    for all gadget ∈ gadgets do
        new_chain ← chain + gadget
        regset ← controlled_registers(new_chain)
        if regset not in regset_to_chain or new_chain is shorter than regset_to_chain[regset] then
            regset_to_chain[regset] ← new_chain
            queue.push(new_chain)
        end if
    end for
end while
```

Salls [37] developed the method described above. Below, we describe a method for generating chain that sets register values to specified values. The rest of chains, such as writing to memory and calling a function, can be obtained by appending just one gadget to the chain that initializes the registers. The method can be divided into three steps:

1. **Gadget summary composing.** Symbolic execution [92–94] of each gadget instructions occurs. Gadget summaries are composed through a static analysis of the resulting SMT expressions and queries to the SMT solver.

2. **Chaining gadgets.** At this step, one searches for the shortest chains to initialize arbitrary sets of registers. Dijkstra's algorithm [112] inspired the proposed algorithm 5 for finding the shortest paths from one of the graph vertices to all the others. An empty mapping is created from the register sets to the shortest chains that initialize these registers. An empty chain is added to the queue. The algorithm takes the chains from the queue. For each gadget, a new chain is created by appending this gadget to the chain taken from the queue. The set of registers initialized by a new chain (*controlled_registers*) is calculated. If there is no such a set in the mapping, or the resulting chain is shorter than the one in the mapping, then a new chain is added into the mapping for this set. Also, the same chain is added to the queue. Thus, mapping is obtained from the sets of registers into the shortest chains that initialize these registers.

3. **Placing a ROP chain on stack.** Symbolic execution of the entire ROP chain starts. Free symbolic variables are created for values loaded from the stack. At the end of the symbolic execution process, a conjunction of the equalities of the requested registers to the given values is constructed. As a result of solving this conjunction, the SMT solver provides bytes that are to be placed on stack.

The described method, unlike the previous one, allows one to use those gadgets in chains that initialize several registers at once, as well as gadgets that perform arithmetic operations on registers loaded by other gadgets (SMT solver calculates the correct values on stack). Moreover, this method allows for the shortest chains to be selected.

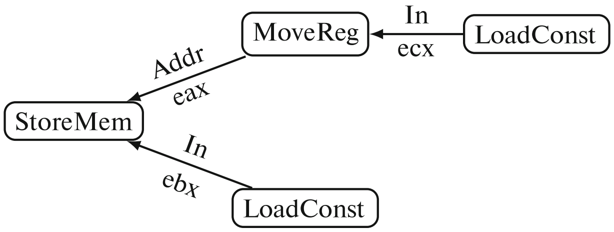


Fig. 8. Gadget tree that stores arbitrary value at arbitrary memory address.

Similar to the method of Follner et al. [29], chains are generated by an open-source tool Exrop [44] that builds gadget summaries as a result of their instructions symbolic execution through the Triton framework [113]. Suitable gadgets are selected for each register to be set. An SMT solver checks postcondition satisfiability. It is worth noticing that the tool supports jump-oriented (JOP) gadgets similar to ROPium (Section 3).

15.6. Semantic Tree Generation

Schwartz et al. [6] propose an approach to the ROP chains generation based on semantic trees. The authors created the QooL language for writing ROP chains, which is not Turing-complete but allows expressing ROP chains used in practice (a library function call, system call, and writing to memory). The process of translating a QooL program into a ROP chain consists of the following steps:

1. Generation of semantic trees by tiling [114] the original QooL program abstract syntax tree. The semantic tree consists of abstract gadgets (gadget types) that define an instruction set architecture and are described in Section 1.

2. Real gadgets (found in the program) assignment to abstract gadgets from the semantic tree. An example of a real gadgets tree is given in Figure 8. Gadget types are stored in tree nodes. Type parameter names and their values (specific registers) are stored on edges. The gadget tree writes an arbitrary value to an arbitrary memory address. The stored value and the address are loaded from the stack into the *ebx* and *ecx* registers, respectively. The address from the *ecx* register is moved to the *eax* register. After that, the value of the *ebx* register is stored at address *eax*.

3. Scheduling a gadgets tree and generating a ROP chain.

The first step is the lazy generation of all possible semantic trees from abstract gadgets. It is necessary because some gadgets may not be available in a particular program. In the second step, gadgets are assigned to each semantic tree. If it is not possible to assign a specific gadget to each abstract gadget, then the semantic tree is discarded, and the following is taken. In case of successful assignment, the tree of real gadgets is passed to the third step. To generate the ROP chain, the gadget tree must be linearized, i.e., scheduled. Gadgets tree scheduling should consider: data dependencies between the gadget registers and "clobbered" registers. It means the following (Fig. 9):

1. The schedule should satisfy the topological order of the tree.

2. If the output register of gadget a is used by gadget b, then this register should not be "clobbered" by any gadget in the schedule between a and b.

During semantic trees generation, certain gadget types absence possibility is considered, and all available rules are applied sequentially to express the vertices of the abstract syntax tree through semantic trees from abstract gadgets. For example, the authors noticed that the ROP chain generation success increases if the following rule for expressing a vertex, that stores a value in memory, is added:

1. `mov [eax], 0 ; ret`
2. `pop ebx ; ret`
3. `add [eax], ebx ; ret`

Ouyang et al. [28] extended the QooL instruction set to a Turing-complete set. In general, they repeat the approach of Schwartz et al. [6] with the construction of semantic trees, using a value liveness analysis when dealing with side effects. It is worth noticing that there are attempts to implement the method of Schwartz et al. that have open-source code [39, 41, 115].

16. RESTRICTED SYMBOLS

Automatic ROP chain generation tools should deal with input sanitization for a particular exploit. For example, data copied through strcpy function cannot contain null bytes. Both gadget addresses and values loaded by gadgets from the stack can contain restricted symbols.

In simplest case, gadget addresses are sanitized by dropping gadgets that contain restricted symbol in the address. Many tools act the same. However, this approach inevitably leads to a reduction in the gadget catalog, which causes a lack of gadgets and the need to combine them to model missing gadgets.

The situation is much more complicated when restricted symbols are in data to be loaded onto registers (values of function arguments and values necessary for writing to memory). To solve this problem, one can use various arithmetic operations to obtain values containing restricted symbols.

A detailed description of dealing with restricted symbols is given in an article by Ding et al. [27]. It is worth noting that the authors are restricting all non-printable characters, which may be excessive in some cases. However, their methods are applicable in the more general case of an arbitrary set of restricted symbols. For each gadget, the authors construct a semantic tree that describes the gadget functionality and

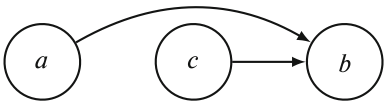


Fig. 9. Scheduling gadget tree.

contains explicit dependencies between registers and memory regarding arithmetic operations and memory interaction operations.

The constructed semantic trees help to build a finite state machine (Fig. 10). This FSM is intended to search for instructions that load a value into a register. The vertices in the finite state machine are the following states corresponding to different loaded values:

- Z – zero,
- SI – small number,
- GC – number containing no restricted symbols,
- BC – number containing restricted symbols,
- T – final state.

Between these vertices, there are edges corresponding to specific gadgets, provided by the gadget catalog. Possible transitions between states of a finite state machine are the following:

1. $SI \to T$, the edge from the vertex with a small number to the final state corresponds to the gadget with the instruction directly initializing this value in the register.

2. $GC \to T$, an edge with a pop gadget leads from a vertex with a number that does not contain restricted symbols to the final state.

3. $Z \to T$, an edge with xor instruction leads from a vertex with zero to the final state.

4. $SI \leftrightarrow Z$, edges with instructions inc, dec lead from number to zero and vice versa

5. $Z \to GC$, an edge with arithmetic instructions and, or, sal, shl, shr, sar leads from the vertex with zero to a state with a number that does not contain restricted symbols.

6. $BC \to GC$, edges consisting of a combination of two arithmetic operations, for example, $a + b - c$, lead from a vertex with a number containing restricted symbols to a vertex that does not contain restricted symbols.

The algorithm starts from state corresponding to the value that needs to be set in a particular register. By traversing the states of this state machine, one can solve the problem of possible ROP chain data sanitization. The algorithm is interrupted if (1) the final state is reached, which corresponds to the successful finding of a combination of gadgets that solve the task, or (2) there are no transitions to other states from the current one.

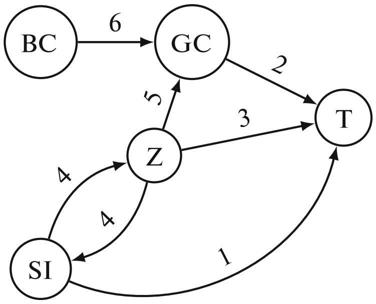


Fig. 10. State machine describing the input sanitizing algorithm.

17. EXPERIMENTAL TOOLS COMPARISON

We performed experimental testing of tools, that have an available source code, with rop-benchmark [45] test system. This system provides a reproducible environment for checking generation success and exploitability of ROP chains invoking `execve ("/bin/sh", 0, 0)` system call. The testing system supports Linux x86-64 platform. We took executable files and libraries from minimal installations of several popular distributions: CentOS 7, Debian 10, OpenBSD 6.2, OpenBSD 6.4. We considered OpenBSD 6.2 and 6.4 because the authors of this operating system intentionally reduced the number of ROP gadgets [116].

Table 6 presents experimental evaluation results. Four columns correspond to four sets of test files. The first line shows the total number of test files in each set. The second line – the number of binaries that contain syscall gadgets. The third line – the number of files that have a working ROP chain created by at least one tool. Below are the lines with the tools, and next to each tool the following information is indicated:

- OK – the number of test files for which the created ROP chain is exploitable, i.e., leads to the opening of the system shell.
- F – the number of test files for which the created ROP chain is not workable, i.e., for some reason, it does not open the system shell. It is worth noticing that we performed 10 runs of an executable file. If at least one did not lead to shell opening, the generated chain is not considered workable.
- TL – the number of test files on which the tool runtime exceeded the set limit of 1 hour.

Ropper almost always generates a ROP chain script file, so the F number was not evaluated.

The experimental comparison included only publicly available tools that could fully automatically generate a ROP chain which performs a system call for x86-64 architecture on Linux family operating system. We did not consider the mona.py tool [34] due to the operating system. Others can work only with x86 architecture (32-bit) [42], ARM [33]. Some available tools failed to be successfully integrated into the auto-

Table 6. The experimental evaluation of automatic ROP chain generation tools.

Test suite	Synthetic			OpenBSD			OpenBSD			Debian			CentOS		
Number of files	22			410			397			689			649		
Has syscall gadget	21			98			87			139			121		
At least one OK	13			19			50			115			72		
Tool	OK	F	TL	OK	F	TL	OK	F	TL	OK	F	TL	OK	F	TL
ROPgadget [35]	1	0	0	2	0	0	4	0	0	7	0	0	8	0	0
Ropper [38]	2	–	0	3	–	0	15	–	0	53	–	0	31	–	0
Exrop [44]	9	0	0	0	33	28	11	27	13	76	19	5	48	8	12
Angrop [37]	8	1	0	10	1	2	25	2	3	86	12	1	54	9	0
ROPium [36]	10	1	0	18	4	0	43	6	1	103	10	0	64	11	0

mated execution system [39]. The test system does not include tests with restricted symbols (for instance, 0x00 in case of overflow while copying via `strcpy`) because no tool completely supports them, i.e., checks the presence of restricted symbols not only in the gadget addresses but also in gadget parameters values on stack. Tools ROPium [36], Ropper [38] can only restrict symbols in gadget addresses.

18. CONCLUSIONS

This article provides a detailed overview of code reuse attacks and automated exploit generation techniques for such attacks. Code reuse attacks imply the use of code pieces from the program address space, the pieces are called *gadgets*. Gadgets are linked in a chain that performs a malicious payload. We divide the process of code reuse exploits generation into four stages: searching for gadgets in an exploitable program, determining gadgets semantics, combining gadgets in chains, and generating input data exploiting the vulnerability. In the first stage, found gadgets constitute a gadget catalog. After that, one derives the gadgets semantics and catalog them. There are three ways to present gadget semantics: parameterized semantic types, gadget summaries, gadget dependency graphs. In the third stage, gadgets can be chained both by searching according to regular expression templates or considering their semantics.

In some cases, if the set of gadgets in the catalog is Turing-complete, then the gadgets can be used as the target architecture for a compiler. Moreover, some approaches construct ROP chains with genetic algorithms, while others use SMT solvers. It is worth noting that methods for automated generation of chains that exploit data flow (DOP) are beyond the article.

We propose the ROP Benchmark [45] for experimental comparison of ROP chain generation tools. We compared open-source ROP chain generation tools on Linux x86-64 platform. In particular, the comparison was carried out on OpenBSD distributions, the authors of which intentionally reduce the number of ROP gadgets [116].

During chains generation, it is crucial to take account of restricted symbols. For example, if `strcpy` function processes input data, they cannot contain zero bytes. However, just a few authors consider restricted symbols in the chain generation methods.

There are a wide range of code-reuse attack methods (ROP, JOP, and others). The question of what set of such methods appears to be enough to implement exploitation is still open. For example, an attacker could manually form a JOP chain for some vulnerable program, while advanced methods allowed generating a regular ROP chain for the same program. The question arises whether it is possible to improve chain generation methods without use of complicated code-reuse attacks.

A promising direction for research is investigating how to bypass address space randomizing protections without information leakage and brute-force.

Most methods do not use gadgets that have arbitrary memory dereference as a side-effect [29]. Their consideration would expand the gadget catalog and improve chain generation methods.

FUNDING

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

REFERENCES

1. The Heartbleed bug. http://heartbleed.com.
2. Halperin, D., Heydt-Benjamin, T.S., Ransford, B., Clark, S.S., Defend, B., Morgan, W., Fu, K., Kohno, T., and Maisel, W.H., Pacemakers and implantable cardiac defibrillators: software radio attacks and zero-power defenses, *Proc. 2008 IEEE Symp. on Security and Privacy (sp 2008)*, Red Hook, NY: Curran Assoc., 2008, pp. 129–142.
https://doi.org/10.1109/SP.2008.31
3. CWE-2020, CWE top 25 most dangerous software weaknesses.
https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.
4. Peslyak, A., Getting around non-executable stack (and fix), 1997. https://seclists.org/bugtraq/1997/Aug/63.
5. Shacham, H., The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), *Proc. 14th ACM Conf. on Computer and Communications Security (CCS '07), Alexandria, Virginia, USA*, New York, NY: Assoc. Comput. Mach., 2007, pp. 552–561.
https://doi.org/10.1145/1315245.1315313
6. Schwartz, E.J., Avgerinos, T., and Brumley, D., Q: Exploit hardening made easy, *Proc. 20th USENIX Conf. on Security (SEC'11), San Francisco, CA*, Berkeley, CA:USENIX Assoc., 2011. https://www.usenix.org/legacy/event/sec11/tech/full_papers/Schwartz.pdf.
7. Roemer, R., Buchanan, E., Shacham, H., and Savage, S., Return-oriented programming: Systems, languages, and applications, *ACM Trans. Inf. Syst. Secur.*, 2012, vol. **15**, no. 1, 1–36.
https://doi.org/10.1145/2133375.2133377
8. Kornau, T., Return oriented programming for the ARM architecture, *MSc Thesis*, Bochum: Ruhr-Univ., 2009. https://bxi.es/Reversing-Exploiting/ROP/Return%20Oriented%20Programming%20for%20ARM.pdf.
9. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M., Return-oriented programming without returns, *Proc. 17th ACM Conf. on Computer and Communications Security (CCS'10), Chicago, IL*, New York, NY: Assoc. Comput. Mach., 2010, pp. 559–572.
https://doi.org/10.1145/1866307.1866370
10. Davi, L., Dmitrienko, A., Sadeghi, A.-R., and Winandy, M., *Return-Oriented Programming without Returns on ARM: Technical Report HGI-TR-2010-002*, Bochum: Horst Gortz Inst. IT Secur., 2010. https://www.ei.ruhr-uni-bochum.de/media/trust/veroeffentlichungen/2010/07/21/ROP-without-Returns-on-ARM.pdf.
11. Huang, Z.-S. and Harris, I.G., Return-oriented vulnerabilities in ARM executables, *Proc. 2012 IEEE Conf. on Technologies for Homeland Security (HST)*, Red Hook, NY: Curran Assoc., 2012, pp. 1–6.
https://doi.org/10.1109/THS.2012.6459817
12. Fraser, O.L., Zincir-Heywood, N., Heywood, M., and Jacobs, J.T., Return-oriented programme evolution with ROPER: a proof of concept, *Proc. Genetic and Evolutionary Computation Conf. Companion (GECCO '17)*, New York, NY: Assoc. Comput. Mach., 2017, pp. 1447–1454.
https://doi.org/10.1145/3067695.3082508
13. Buchanan, E., Roemer, R., Shacham, H., and Savage, S., When good instructions go bad: Generalizing return-oriented programming to RISC, *Proc. 15th ACM Conf. on Computer and Communications Security (CCS'08), Alexandria, Virginia, USA*, New York, NY: Assoc. Comput. Mach., 2008, pp. 27–38.
https://doi.org/10.1145/1455770.1455776
14. Francillon, A. and Castelluccia, C., Code injection attacks on harvard-architecture devices, *Proc. 15th ACM Conf. on Computer and Communications Security (CCS'08), Alexandria, Virginia, USA*, New York, NY: Assoc. Comput. Mach., 2008, pp. 15–26.
https://doi.org/10.1145/1455770.1455775.
15. Lindner, F., Cisco IOS router exploitation, in Black Hat, 2009. https://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-Router-Exploit-PAPER.pdf.
16. Checkoway, S., Feldman, A.J., Kantor, B., Halderman, J.A., Felten, E.W., and Shacham, H., Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage, *Proc. 2009 Conf. on Electronic Voting Technology/Workshop on Trustworthy Elections (EVT/WOTE'09), Montreal, Canada*, Berkeley, CA: USENIX Assoc., 2009.
https://www.usenix.org/legacy/event/evtwote09/tech/full_papers/checkoway.pdf.
17. Bletsch, T., Jiang, X., Freeh, V.W., and Liang, Z., Jump-oriented programming: a new class of code-reuse attack, *Proc. 6th ACM Symp. on Information, Computer and Communications Security (ASIACCS '11), Hong Kong, China*, New York, NY: Assoc. Comput. Mach., 2011, pp. 30–40.
https://doi.org/10.1145/1966913.1966919
18. Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., and Yin, X., Automatic construction of jump-oriented programming shellcode (on the x86), *Proc. 6th ACM Symp. on Information, Computer and Communications Security (ASIACCS'11), Hong Kong, China*, New York, NY: Assoc. Comput. Mach., 2011, pp. 20–29.
https://doi.org/10.1145/1966913.1966918
19. Sadeghi, A., Niksefat, S., and Rostamipour, M., Pure-call oriented programming (PCOP): chaining the gadgets using call instructions, *J. Comput. Virol. Hacking Techn.*, 2018, vol. **14**, no. 2, pp. 139–156.
https://doi.org/10.1007/s11416-017-0299-1
20. Lu, K., Xiong, S., and Gao, D., RopSteg: Program steganography with return oriented programming, *Proc. 4th ACM Conf. on Data and Application Security and Privacy (CODASPY'14)*, New York, NY: Assoc. Comput. Mach., 2014, pp. 265–272.
https://doi.org/10.1145/2557547.2557572
21. Ntantogian, C., Poulios, G., Karopoulos, G., and Xenakis, C., Transforming malicious code to ROP gadgets for antivirus evasion, *IET Inf. Secur.*, 2019, vol. **13**, no. 6, pp. 570–578.
https://doi.org/10.1049/iet-ifs.2018.5386
22. Mu, D., Guo, J., Ding, W., Wang, Z., Mao, B., and Shi, L., ROPOB: obfuscating binary code via return oriented programming, in *Security and Privacy in Communication Networks*, Cham: Springer-Verlag, 2018,

pp. 721–737.
https://doi.org/10.1007/978-3-319-78813-5_38

23. Bosman, E. and Bos, H., Framing signals—a return to portable shellcode, *Proc. 2014 IEEE Symp. on Security and Privacy*, Red Hook, NY: Curran Assoc., 2014, pp. 243–258.
https://doi.org/10.1109/SP.2014.23

24. Borrello, P., Coppa, E., D'Elia, D.C., and Demetrescu, C., The ROP needle: hiding trigger-based injection vectors via code reuse, *Proc. 34th ACM/SIGAPP Symp. on Applied Computing (SAC'19), Limassol, Cyprus*, New York, NY: Assoc. Comput. Mach., 2019, pp. 1962–1970.
https://doi.org/10.1145/3297280.3297472

25. Hund, R., Holz, T., and Freiling, F.C., Return-oriented rootkits: bypassing kernel code integrity protection mechanisms, *Proc. 18th Conf. on USENIX Security Symp. (SSYM'09), Montreal, Canada*, Berkeley, CA: USENIX Assoc., 2009, pp. 383–398.
https://www.usenix.org/legacy/events/sec09/tech/full_papers/hund.pdf.

26. Quynh, N.A., OptiROP: hunting for ROP gadgets in style, 2013.
https://media.blackhat.com/us-13/US-13-Quynh-OptiROP-Hunting-for-ROP-Gadgets-in-Style-Slides.pdf.

27. Ding, W., Xing, X., Chen, P., Xin, Z., and Mao, B., Automatic construction of printable return-oriented programming payload, *Proc. 2014 9th Int. Conf. on Malicious and Unwanted Software "The Americas" (MALWARE)*, Washington, DC: IEEE Comput. Soc., 2014, pp. 18–25.
https://doi.org/10. 1109/MALWARE.2014.6999408.

28. Y. Ouyang, Q. Wang, J. Peng, and J. Zeng, An advanced automatic construction method of ROP, *Wuhan Univ. J. Nat. Sci.*, 2015, vol. **20**, no. 2, pp. 119–128.
https://doi.org/10.1007/s11859-015-1069-x

29. Follner, A., Bartel, A., Peng, H., Chang, Y.-C., Ispoglou, K., Payer, M., and Bodden, E., PSHAPE: automatically combining gadgets for arbitrary method execution, *Proc. 12th Int. Workshop Security and Trust Management (STM'2016), Heraklion, Crete, Greece, September 26–27, 2016*, Cham: Springer-Verlag, 2016, pp. 212–228.
https://doi.org/10.1007/978-3-319-46598-2_15.

30. Milanov, B., ROPGenerator: practical automated ROP-chain generation, 2018. https://youtu.be/rz7Z9fBLVs0.

31. Mosier, N. and Johnson, P., ROP with a 2nd stack, 2019. http://www.cs.middlebury.edu/~nmosier/portfolio/rsrc/ropc-slides.pdf.

32. Follner, A., Bartel, A., Peng, H., Chang, Y.-C., Ispoglou, K., Payer, M., and Bodden, E., PSHAPE-practical support for half-automated program exploitation. https://github.com/Alexandre-Bartel/inspector-gadget.

33. Fraser, O.L., ROPER: a genetic ROP-chain development tool. https://github.com/oblivia-simplex/roper.

34. mona.py, Corelan Consulting BVBA.
https://github.com/corelan/mona.

35. Salwan, J., ROPgadget tool.
https://github.com/JonathanSalwan/ROPgadget.

36. Milanov, B., ROPium.
https://github.com/Boyan-MILANOV/ropium.

37. Salls, C., angrop. https: //github.com/salls/angrop.

38. Schirra, S., Ropper. https://github.com/sashs/ropper.

39. Paul, ROPC. https://github.com/pakt/ropc.

40. ropc-llvm, Program STIC.
https://github.com/programa-stic/ropc-llvm.

41. Stewart, J., An open source, multiarchitecture ROP compiler. https://github.com/jeffball55/rop_compiler.

42. SQLab, SQLab ROP payload generation.
https://github.com/SQLab/ropchain.

43. Gautham, A.V. and Singh, S., ROPilicious.
https://github.com/ROPilicious/src.

44. d4em0n, Exrop. https: //github.com/d4em0n/exrop.

45. Nurmukhametov, A. and Vishnyakov, A., ROP benchmark. https://github.com/ispras/rop-benchmark.

46. W^X now mandatory in OpenBSD, 2016.
https://undeadly.org/cgi?action=article&sid=20160527203200.

47. A detailed description of the data execution prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC edition 2005, and Windows Server 2003. https://support.microsoft.com/kb/875352/EN-US/.

48. van de Ven, A., New security enhancements in Red Hat Enterprise Linux v.3, update 3, 2004.
https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.

49. Tanenbaum, A.S. and Bos, H., Buffer overflow attacks, in *Modern Operating Systems*, 4th ed., London: Pearson, 2015, ch. 9.6.1, pp. 640–649. ISBN 978-0133591620

50. CWE-123: Write-what-where condition. https://cwe.mitre.org/data/definitions/123.html.

51. Spengler, B., PaX: The guaranteed end of arbitrary code execution. https://grsecurity.net/PaX-presentation.pdf.

52. Bhatkar, S., DuVarney, D.C., and Sekar, R., Address obfuscation: An efficient approach to combat a broad range of memory error exploits, *Proc. 12th USENIX Security Symp., Washington, DC, August 4–8, 2003*, Berkeley, CA: USENIX Assoc., 2003, vol. 12, no. 2, pp. 291–301.
https://www.usenix.org/legacy/event/sec03/tech/full_papers/bhatkar/bhatkar.pdf.

53. Fedotov, A.N., Padaryan, V.A., Kaushan, V.V., Kurmangaleev, S.F., Vishnyakov, A.V., and Nurmukhametov, A.R., Software defect severity estimation in presence of modern defense mechanisms, *Proc. Inst. Syst. Program., Russ. Acad. Sci.*, 2016, vol. **28**, no. 5, pp. 73–92.
https://doi.org/10.15514/ISPRAS-2016-28(5)-4

54. Procedure Linkage Table (processor-specific). https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-1235.html.

55. 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.

56. Dullien, T.F., Weird machines, exploitability, and provable unexploitability, *IEEE Trans. Emerg. Topics Comput.*, 2018, vol. **8**, no. 2, pp. 391–403.
https://doi.org/10.1109/TETC.2017.2785299

57. Graziano, M., Balzarotti, D., and Zidouemba, A., ROPMEMU: A framework for the analysis of complex code-reuse attacks, *Proc. 11th ACM on Asia Conf. on Computer and Communications Security (ASIA CCS'16)*,

New York, NY: Assoc. Comput. Mach., 2016, pp. 47–58. https://doi.org/10.1145/2897845.2897894

58. Schwartz, E.J., Avgerinos, T., and Brumley, D., Update on Q: Exploit hardening made easy, 2012. https://edmcman.github.io/papers/usenix11-update.pdf.

59. Weidler, N.R., Brown, D., Mitchell, S.A., Anderson, J., Williams, J.R., Costley, A., Kunz, C., Wilkinson, C., Wehbe, R., and Gerdes, R., Return-oriented programming on a resource constrained device, *Sustainable Comput.: Inf. Syst.*, 2019, vol. 22, pp. 244–256. ISSN 2210-5379. https://doi.org/10.1016/j.suscom.2018.10.002

60. Vishnyakov, A.V., Nurmukhametov, A.R., Kurmangaleev, S.F., and Gaisaryan, S.S., A method for analyzing code-reuse attacks, *Program. Comput. Software*, 2019, vol. **45**, pp. 473–484. https://doi.org/10.1134/S0361768819080061

61. Roglia, G.F., Martignoni, L., Paleari, R., and Bruschi, D., Surgically returning to randomized lib(c), *Proc. 2009 Annual Computer Security Applications Conf.*, Red Hook, NY: Curran Assoc., 2009, pp. 60–69. https://doi.org/10.1109/ACSAC.2009.16

62. PE format-Import Address Table. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#import-address-table.

63. ld.so(8)-Linux manual page. http://man7.org/linux/man-pages/man8/ld.so.8.html.

64. Kirsch, J., Bierbaumer, B., Kittel, T., and Eckert, C., Dynamic loader oriented programming on Linux, *Proc. 1st Reversing and Offensive-oriented Trends Symp. (ROOTS), Vienna, Austria*, New York, NY: Assoc. Comput. Mach., 2017, pp. 1–13. https://doi.org/10.1145/3150376.3150381

65. Ward, B.C., Skowyra, R., Spensky, C., Martin, J., and Okhravi, H., The leakage-resilience dilemma, *Proc. 24th European Symp. on Research in Computer Security "Computer Security—ESORICS 2019," Luxembourg, September 23–27, 2019*, Cham: Springer-Verlag, 2019, pp. 87–106. https://doi.org/10.1007/978-3-030-29959-0_5

66. Dai Zovi, D., Practical return-oriented programming, SOURCE Boston, 2010. https://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf.

67. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H., Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks, *Proc. 7th USENIX Security Symp. San Antonio, Texas, January 26–29, 1998*, Berkeley, CA: USENIX Assoc., 1998, vol. 98, pp. 63–78. https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf.

68. mprotect(2)-Linux manual page. http://man7.org/linux/manpages/man2/mprotect.2.html.

69. VirtualProtect function. https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualprotect.

70. van Eeckhoutte, P., Exploit writing tutorial part 10: chaining DEP with ROP—the Rubik's™ cube, 2010. https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorialpart-10-chaining-dep-with-rop-therubikstm-cube/.

71. Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., and Boneh, D., Hacking blind, *Proc. 2014 IEEE Symp. on Security and Privacy (SP'14)*, Washington, DC: IEEE Comput. Soc., 2014, pp. 227–242. https://doi.org/10.1109/SP.2014.22

72. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R., Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization, *Proc. 2013 IEEE Symp. on Security and Privacy (SP'13)*, Washington, DC: IEEE Comput. Soc., 2013, pp. 574–588. https://doi.org/10.1109/SP.2013.45

73. Nurmukhametov, A.R., Zhabotinskiy, E.A., Kurmangaleev, S.F., Gaissaryan, S.S., and 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

74. Göktas, E., Kollenda, B., Koppe, P., Bosman, E., Portokalidis, G., Holz, T., Bos, H., and Giuffrida, C., Position-independent code reuse: On the effectiveness of ASLR in the absence of information disclosure, *Proc. 2018 IEEE European Symp. on Security and Privacy (EuroSP)*, Washington, DC: IEEE Comput. Soc., 2018, pp. 227–242. https://doi.org/10.1109/EuroSP.2018.00024

75. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M., Control-flow integrity: precision, security, and performance, *ACM Comput. Surv.*, 2017, vol. 50, no. 1, pp. 16:1–16:33. https://doi.org/10.1145/3054924

76. Carlini, N. and Wagner, D., ROP is still dangerous: breaking modern defenses, *Proc. 23rd USENIX Security Symp., San Diego, CA, August 20–22, 2014*, Berkeley, CA: USENIX Assoc., 2014, pp. 385–399. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-carlini.pdf.

77. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P., On the expressiveness of return-into-libc attacks, *Proc. 14th Int. Symp. Recent Advances in Intrusion Detection (RAID 2011), Menlo Park, CA, USA, September 20–21, 2011*, Berlin: Springer-Verlag, 2011, pp. 121–141. https://doi.org/10.1007/978-3-642-23644-0_7

78. Lan, B., Li, Y., Sun, H., Su, C., Liu, Y., and Zeng, Q., Loop-oriented programming: a new code reuse attack to bypass modern defenses, *Proc. 2015 IEEE Trustcom/BigDataSE/ISPA (TRUSTCOM'15)*, Washington, DC: IEEE Comput. Soc., 2015, vol. **1**, pp. 190–197. https://doi.org/10.1109/Trustcom.2015.374

79. Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F., Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection, *Proc. 23rd USENIX Security Symp., San Diego, CA, August 20–22, 2014*, Berkeley, CA: USENIX Assoc., 2014, pp. 401–416. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-davi.pdf.

80. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T., Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in C++ applications, *Proc. 2015 IEEE*

Symp. on Security and Privacy (SP'15), Washington, DC: IEEE Comput. Soc., 2015, pp. 745–762. https://doi.org/10.1109/SP.2015.51

81. Wang, C., Chen, B., Liu, Y., and Wu, H., Layered object-oriented programming: advanced VTable reuse attacks on binary-level defense, *IEEE Trans. Inf. Forensics Secur.*, vol. **14**, no. 3, pp. 693–708, 2019. https://doi.org/10.1109/TIFS.2018.2855648
82. Guo, Y., Chen, L., and Shi, G., Function-oriented programming: a new class of code reuse attack in C applications, *Proc. 6th Annual IEEE CNS Conf. on Communications and Network Security*, Piscataway, NJ: Inst. Electr. Electron. Eng., 2018, pp. 1–9. https://doi.org/10.1109/CNS.2018.8433189
83. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., and Iyer, R.K., Non-control-data attacks are realistic threats, *Proc. 14th Conf. on USENIX Security Symp. (SSYM'05)*, Berkeley, CA: USENIX Assoc., 2005, **vol. 14**, pp. 177–192. http://static.usenix.org/event/sec05/tech/full_papers/chen/chen.pdf.
84. Hu, H., Chua, Z.L., Adrian, S., Saxena, P., and Liang, Z., Automatic generation of dataoriented exploits, *Proc. 24th USENIX Conf. on Security Symp. (SEC'15)*, Berkeley, CA: USENIX Assoc., 2015, pp. 177–192. https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-hu.pdf.
85. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., and Liang, Z., Data-oriented programming: on the expressiveness of noncontrol data attacks, *Proc. 2016 IEEE Symp. on Security and Privacy (SP'16)*, Washington, DC: IEEE Comput. Soc., 2016, pp. 969–986. https://doi.org/10.1109/SP.2016.62.
86. Pewny, J., Koppe, P., and Holz, T., STEROIDS for DOPed applications: a compiler for automated data-oriented programming, *Proc. 2019 IEEE European Symp. on Security and Privacy Workshops (EuroS&PW)*, Red Hook, NY: Curran Assoc., 2019, pp. 111–126. https://doi.org/10.1109/EuroSP.2019.00018.
87. Ispoglou, K.K., AlBassam, B., Jaeger, T., and Payer, M., Block oriented programming: Automating data-only attacks, *Proc. 2018 ACM SIGSAC Conf. on Computer and Communications Security (CCS'18), Toronto, Canada*, New York, NY: Assoc. Comput. Mach., 2018, pp. 1868–1882. https://doi.org/10.1145/3243734.3243739
88. Avgerinos, T., Cha, S.K., Hao, B.L.T., and Brumley, D., AEG: automatic exploit generation, *Commun. ACM*, 2011, vol. **57**, no. 2, pp. 283–300. http://security.ece.cmu.edu/aeg/aeg-current.pdf.
89. Cha, S.K., Avgerinos, T., Rebert, A., and Brumley, D., Unleashing Mayhem on binary code, *Proc. 2012 IEEE Symp. on Security and Privacy*, Red Hook, NY: Curran Assoc., 2012, pp. 380–394. https://doi.org/10.1109/SP.2012.31
90. Padaryan, V.A., Kaushan, V.V., and Fedotov, A.N., Automated exploit generation for stack buffer overflow vulnerabilities, *Program. Comput. Software*, 2015, vol. **41**, no. 6, pp. 373–380. https://doi.org/10.1134/S0361768815060055
91. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser C., Kruegel, C., and Vigna, G., SOK: (state of) the art of war: Offensive techniques in binary analysis, *Proc. 2016 IEEE Symp. on Security and Privacy (SP)*, Red Hook, NY: Curran Assoc., 2016, pp. 138–157. https://doi.org/10.1109/SP. 2016.17
92. King, J.C., Symbolic execution and program testing, *Commun. ACM*, 1976, vol. **19**, no. 7, pp. 385–394. https://doi.org/10.1145/360248.360252
93. Schwartz, E.J., Avgerinos, T., and Brumley, D., All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), *Proc. 2010 IEEE Symp. on Security and Privacy*, Red Hook, NY: Curran Assoc., 2010, pp. 317–331. https://doi.org/10.1109/SP.2010.26
94. Godefroid, P., Levin, M.Y., and Molnar, D.A., Automated whitebox fuzz testing, *Proc. 15th Network and Distributed System Security Symp. (NDSS'2008), San Diego, California, USA*, Reston, VA: Internet Soc., 2008, **vol. 8**, pp. 151–166. https://www.microsoft.com/en-us/research/publication/automatedwhitebox-fuzz-testing/.
95. Homescu, A., Stewart, M., Larsen, P., Brunthaler, S., and Franz, M., Microgadgets: size does matter in turing-complete returnoriented programming, *Proc. 6th USENIX Workshop on Offensive Technologies (WOOT'12)*, Berkeley, CA: USENIX Assoc., 2012. https://www.usenix.org/system/files/conference/woot12/woot12-final9.pdf.
96. BARF: binary analysis and reverse engineering framework, Program STIC. https://github.com/programa-stic/barf-project.
97. Heffner, C., MIPS ROP IDA plug-in. https://github.com/devttys0/ida/tree/master/plugins/mipsrop.
98. Mosier, N., A pair of return-oriented programming utilities: a gadget finder and ROP compiler. https://github.com/nmosier/rop-tools.
99. lucasg, ROP database plug-in for IDA. https://github.com/lucasg/idarop.
100. Jager, I. and Brumley, D., *Efficient Directionless Weakest Preconditions: Technical Report CMU-CyLab-10-002*, Pittsburgh, PA: Carnegie Mellon Univ., 2010. http://security.ece.cmu.edu/pubs/CMUCyLab10002.pdf.
101. Nethercote, N. and Seward, J., Valgrind: a framework for heavyweight dynamic binary instrumentation, *Proc. 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'07)*, New York, NY: Assoc. Comput. Mach., 2007, pp. 89–100. https://doi.org/10.1145/1273442.1250746
102. Dullien, T. and Porst, S., REIL: a platform-independent intermediate representation of disassembled code for static code analysis, 2009. https://static.googleusercontent.com/media/zynamics.com/en//downloads/csw09.pdf.
103. Padaryan, V.A., Soloviev, M.A., and Kononov, A.I., Modeling operational semantics of machine instructions, *Proc. Inst. Syst. Program., Russ. Acad. Sci.*, 2010, **vol. 19**, pp. 165–186. https://www.ispras.ru/en/proceedings/isp_19_2010/isp_19_2010_165/.
104. Soloviev, M.A., Bakulin, M.G., Gorbachev, M.S., Manushin, D.V., Padaryan, V.A., and. Panasenko, S.S, Next-generation intermediate representations for binary code analysis, *Program. Comput. Software*, 2019, vol. **45**,

pp. 424–437.
https://doi.org/10.1134/S0361768819070107

105. Heitman, C. and Arce, I., BARF: a multiplatform open source binary analysis and reverse engineering framework, *XX Congr. Argentino de Ciencias de la Computación*, San Justo: Univ. Nac. La Matanza, 2014. http://sedici.unlp.edu.ar/handle/10915/42157.

106. Vishnyakov, A.V., Semantic verification of linear machine instruction sequence, 2019.
https://vishnya.xyz/vishnyakov-coursework2019.pdf.

107. Vishnyakov, A., Nurmukhametov, A., Kurmangaleev, S., and Gaisaryan, S., Method for analysis of code-reuse attacks—reverse engineering of ROP exploits. https://vishnya.xyz/vishnyakovisprasopen2018.pdf.

108. Barrett, C., Fontaine, P., and Tinelli, C., *The SMT-LIB Standard: Version 2.6*, Iowa City, IA: Univ. Iowa, 2017. https://www.SMT-LIB.org.

109. Dullien, T., Kornau, T., and Weinmann, R.-P., A framework for automated architecture-independent gadget search, 2010. https://www.usenix.org/legacy/events/woot10/tech/full_papers/Dullien.pdf.

110. Follner, A., Bartel, A., and Bodden, E., Analyzing the gadgets: towards a metric to measure gadget quality, *Proc. 8th Int. Symp. "Engineering Secure Software and Systems" (ESSoS 2016), London, UK, April 6–8, 2016*, Cham: Springer-Verlag, 2016, pp. 155–172.
https://doi.org/10.1007/978-3-319-30806-7_10

111. Fraser, O.L., Urschleim in silicon: return oriented program evolution with ROPER, *MSc Thesis*, Halifax: Dalhouse Univ., 2018.
https://dalspace.library.dal.ca/handle/10222/73879.

112. Dijkstra, E.W., A note on two problems in connexion with graphs, *Num. Math.*, 1959, vol. 1, no. 1, pp. 269–271. ISSN 0945-3245.
https://doi.org/10.1007/BF01386390

113. Saudel, F. and Salwan, J., Triton: a dynamic symbolic execution framework, *Symp. sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'2015)*, Rennes, 2015, pp. 31–54.
https://triton.quarkslab.com/files/sstic2015_slide_en_saudel_salwan.pdf.

114. Aho, A.V., Lam, M.S., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, 2th ed., Boston: Addison-Wesley, 2006, ch. 6, pp. 563–565.

115. Stewart, J. and Dedhia, V., ROP compiler.
https://css.csail.mit.edu/6.858/2015/projects/je25365-ve25411.pdf.

116. Mortimer, T., Removing ROP gadgets from OpenBSD, *Proc. AsiaBSDCon 2018 Conf.*, Bucharest, 2019, pp. 13–21.
https://2019.asiabsdcon.org/proc-body-2019.pdf#page=13.