Chapter 2
Background and Evolution of Code-Reuse Attacks

2.1 General Principle of Control-Flow Attacks

In general, control-flow attacks allow an adversary to subvert the intended execution-flow of a program by exploiting a program error. For instance, a buffer overflow error can be exploited to write data beyond the boundaries of the buffer. As a consequence, an adversary can overwrite critical control-flow information which is located close to the buffer. Since control-flow information guide the program's execution-flow, an adversary can thereby trigger malicious and unintended program actions such as installing a backdoor, injecting a malware, or accessing sensitive data.

Control-flow attacks are performed at application runtime. Hence, they are often referred to as runtime exploits. Note that we use both terms interchangeably in this book. In summary, we define a control-flow attack as follows.

Control-Flow Attack (Runtime Exploit): *A control-flow attack exploits a program error, particularly a memory corruption vulnerability, at application runtime to subvert the intended control-flow of a program. The goal of a control-flow attack is the execution of malicious program actions.*

Loosely speaking, we can distinguish between two major classes of control-flow attacks: (1) code injection and (2) code-reuse attacks. The former class requires the injection of some malicious executable code into the address space of the application. In contrast, code-reuse attacks only leverage benign code already present in the address space of the application. In particular, code-reuse attacks combine small code pieces scattered throughout the address space of the application to generate new malicious program codes on-the-fly.

L. Davi, A.-R. Sadeghi, *Building Secure Defenses Against Code-Reuse Attacks*,
SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-25546-0_2

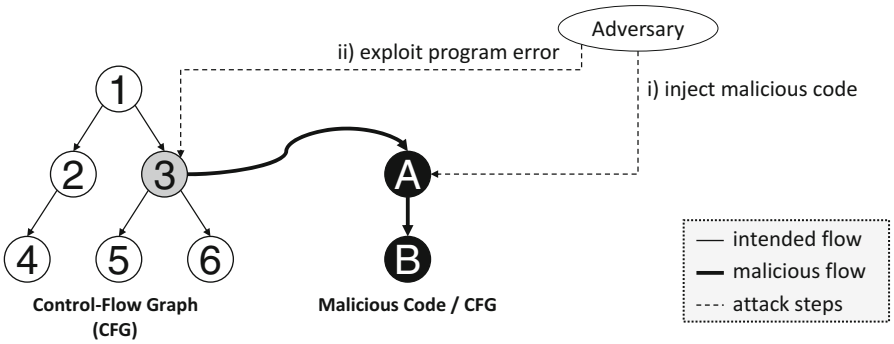

Fig. 2.1 Code injection attacks

A high-level representation of code injection attacks is given in Fig. 2.1. It shows a sample control-flow graph (CFG) with six nodes. The CFG contains all intended execution paths a program might follow at runtime. Typically, the CFG nodes represent the so-called basic blocks (BBLs), where a BBL is a sequence of machine/assembler instructions with a unique entry and exit instruction. The exit instruction can be any branch instruction the processor supports such as direct and indirect jump instructions, direct and indirect call instructions, and function return instructions. The entry instruction of a BBL is an instruction that is target of a branch instruction.

As shown in Fig. 2.1, the CFG nodes are connected via directed edges. These edges represent the possible control-flows, e.g., there is an intended control-flow from node n_3 to n_5 and n_6, where n simply stands for node.

A code injection attack first requires the injection of malicious code. Since programs are allocated into a dedicated memory location at runtime, i.e., the application's virtual address space, the adversary needs to find a free slot to inject her malicious code. Typically, this can be achieved by loading the malicious code into a local buffer that is large enough to hold the entire malicious code. In Fig. 2.1, the malicious code consists of the two nodes n_A and n_B. However, these nodes are not connected to the original CFG. In order to connect the malicious nodes to the intended program nodes, the adversary needs to identify and exploit a program vulnerability. Exploitation of the program vulnerability allows the adversary to tamper with a code pointer, i.e., some control-flow information that guides program execution. A prominent example is a function's return address which is always located on the program stack. Other examples are function pointers or pointers to virtual method tables. In the example exploit shown in Fig. 2.1, n_3 is exploited by the adversary to redirect the execution path to node n_A and n_B. In summary, we define code injection attacks as follows.

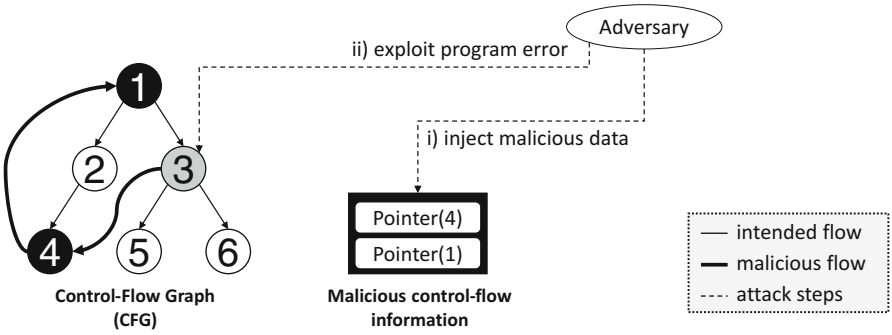

Fig. 2.2 Code-reuse attacks

Code Injection Attack: *A code injection attack is a subclass of control-flow attacks that subverts the intended control-flow of a program to previously injected malicious code.*

Code injection attacks require the injection and execution of malicious code. However, some environments and operating systems deny the execution of code that has just been written into the address space of the program. For instance, a Harvard-based computing architecture strictly separates code and data memory. As a response, a new control-flow attack emerged that only reuses existing code. The high-level principle of these so-called code-reuse attacks is shown in Fig. 2.2.

In contrast to code injection attacks, the adversary only injects malicious data into the address space of the application. Specifically, in the example shown in Fig. 2.2, the adversary injects two code pointers; namely pointers to n_4 and n_1. At the time the adversary exploits the program vulnerability in n_3, the control-flow is redirected to the code pointers the adversary previously injected. Hence, the code-reuse attack in our example leads to the unintended execution path: $n_3 \rightarrow n_4 \rightarrow n_1$. In summary, we define a code-reuse attack as follows.

Code-Reuse Attack: *A code-reuse attack is a subclass of control-flow attacks that subverts the intended control-flow of a program to invoke an unintended execution path inside the original program code.*

Note that the internal workflow and memory layout of a control-flow attack depend on the kind of vulnerability that is exploited. For better understanding, we describe the technical details of control-flow attacks based on a classic buffer overflow vulnerability on the program's stack. Hence, we briefly recall the basics of

a program's stack memory and the typical stack frame layout on x86. Afterwards, we present the technical details of code injection and code-reuse attacks.

2.2 Program Stack and Stack Frame Elements

A program stack operates as a last-in first-out memory area. In particular, it is used in today's programs to hold local variables, function arguments, intermediate results, and control-flow information to ensure correct invocation and return of subroutines. The stack pointer which is stored in a dedicated processor register plays an important role because it points to the top of the stack. Typically, the stack is controlled by two operations: (1) a `POP` instruction that takes one data word off the stack, and (2) a `PUSH` instruction which performs the reverse operation, i.e., it stores one data word on the top of the stack. Both instructions have direct influence on the stack pointer since they change the top of the stack. That is, for stacks that grow from high memory addresses towards low memory addresses (e.g., x86), the `POP` instruction automatically increments the stack pointer by one memory word (on x86 by 4 Bytes), while the `PUSH` instruction decrements it by one word.

In general, the stack is divided into multiple stack frames. Stack frames are associated at function-level, i.e., for each invoked subroutine one stack frame is allocated. The stack frame has a pre-defined structure for each compiler and underlying processor architecture. An example of a typical x86 stack frame and its elements is shown in Fig. 2.3. The depicted stack frame is referenced by two processor registers: the stack pointer (on x86 `%esp`) and the base pointer register (on x86 `%ebp`). As we already mentioned, the stack pointer always points to the top of the stack. In contrast to the stack pointer, the base pointer is constant and fixed per stack frame: it always points to the saved base pointer. The meaning of the saved base pointer and the other stack frame elements is as follows:

- **Function Arguments**: This field holds the arguments which are loaded on the stack by the calling function.
- **Return Address**: The return address indicates where the execution-flow needs to be redirected to upon function return. On x86, the instruction for calling a function (`CALL`) automatically pushes the return address on the stack, where the return address is the address of the instruction that follows the `CALL`.
- **Saved Base Pointer**: The base pointer of a function is used to reference function arguments and local variables on the stack frame. The function prologue of each subroutine initializes the base pointer. This is achieved in two steps. First, the base pointer of the calling function is pushed onto the stack via `PUSH %ebp`. The base pointer stored onto the stack is then referred to as the saved base pointer. Next, the new base pointer is initialized by loading the current stack pointer value into the base pointer register, e.g., `MOV %ebp,%esp`. The function epilogue reverts these operations by first setting the stack pointer to point to the saved

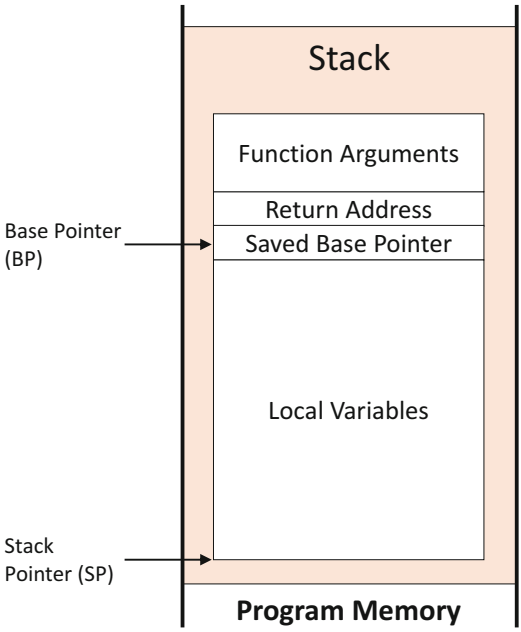

Fig. 2.3 Stack frame memory layout

base pointer field (`MOV %esp,%ebp`), and subsequently loading the saved base pointer to the base pointer register via `POP %ebp`.

- **Local Variables**: The last field of a stack frame holds the local variables such as integer values or local buffers.

2.3 Code Injection

In order to perform a code injection attack, the adversary needs to inject malicious code into the address space of the target application. Typically, this can be achieved by encapsulating the malicious code into a data input variable that gets processed by the application, e.g., a string, file, or network packet.

Figure 2.4 depicts a code injection attack, where a local buffer that can hold up to 100 characters is exploited. The adversary has access to the local buffer, i.e., the application features a user interface from which it expects the user to enter a data input. On x86, data is written from low memory addresses towards high memory addresses, i.e., from the top of the stack towards the saved base pointer in Fig. 2.4.

If the program does not validate the length of the provided data input, it is possible to provide a larger data input than the buffer can actually hold. As a consequence, the stack frame fields which are located above the local buffer are overwritten.

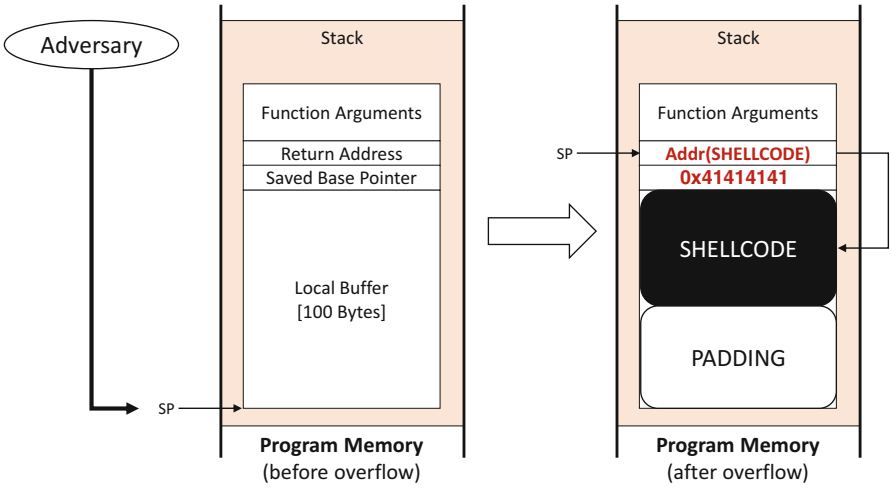

Fig. 2.4 Memory layout of code injection attacks

This can be exploited for the purpose of a code injection attack: the adversary first provides a data input which fills the local buffer with arbitrary pattern bytes and the malicious code. As the main goal of many proof-of-concept exploits is to open a terminal (shell) to the adversary, the malicious code is often referred to as shellcode. Second, the adversary overwrites the saved base pointer with arbitrary bytes (here: $0x41414141$) and replaces the original return address with the runtime address of the shellcode. For systems that do not apply code and data segment randomization, this address is fixed, and can be retrieved by reverse-engineering the program binary using a debugger.

When the subroutine—where the overflow occurred—completed its task and executes its function epilogue instructions, the stack pointer will be reset to the location, where the original return address was stored. As the program is not aware of the overflow, it takes the start address of the shellcode as a return address and redirects execution to the beginning of the shellcode. Thus, the shellcode executes and opens a new terminal to the adversary.

2.4 Data Execution Prevention

One main observation we can make from the code injection attack described above is that malicious code can be encapsulated into a data variable and executed from the program's stack. In fact, code injection attacks were easily possible because data and code got intermixed in memory, and not strictly separated as in Harvard-based processor architectures. Hence, data segments like the stack were marked as readable, writable, and executable (RWX). However, since the main purpose of the

stack is to only hold local variables and control-flow information, we simply need a mechanism to prohibit any code execution from the stack to prevent a code injection attack. As a consequence, kernel patches have been provided to mark the stack as non-executable [34]; e.g., enabled in Solaris 2.6 [3].

The concept of marking the stack as non-executable has been later included into a more general security model referred to as Writable XOR eXecutable ($W \oplus X$) or data execution prevention (DEP) [29]. The main idea of $W \oplus X$ is to prevent any memory page from being writable and executable at the same time. Hence, memory pages belonging to data segments are marked as readable and writable (RW), whereas memory pages that contain executable code are marked as readable and executable (RX). This effectively prevents code injection attacks since an adversary can no longer execute code that has been written via a data variable to a RW-marked memory page. In summary, we define the principle $W \oplus X$ as follows.

Writable xor eXecutable ($W \oplus X$): *The security model of $W \oplus X$ enforces that memory pages are either marked as writable or executable. This prevents a code injection attack, where the adversary first needs to write malicious code into the address space of an application before executing it.*

Today, every mainstream processor architecture features the so-called no-execute bit to facilitate the deployment of $W \oplus X$ in modern operating systems. For instance, Windows-based operating systems enforce DEP since Windows XP SP2 [29].

2.5 Code-Reuse Attacks

After non-executable stacks and $W \oplus X$ have been proposed as countermeasures against control-flow attacks, attackers have instantly demonstrated new techniques to launch control-flow attacks. Instead of injecting malicious code into the address space of the application, an adversary can exploit the benign code which is already present in the address space and marked as executable. Such code-reuse attacks have started as so-called return-into-libc attacks and have been later generalized to return-oriented programming attacks. We describe the technical concepts of both attack techniques in the following.

2.5.1 Return-Into-Libc

The first published exploit that reuses existing code for a return-into-libc attack has been presented by Solar Designer in 1997 [33]. The exploit overwrites the original

return address to point to a critical library function. Specifically, it targets the *system()* function of the standard UNIX C library `libc`, which is linked to nearly every process running on a UNIX-based system. The *system()* function takes as an input a shell command to be executed. For instance, on UNIX-based systems the function call *system("/bin/sh")* opens the terminal program. That said, by invoking the *system()* function, the adversary can conveniently reconstruct the operations of previously injected shellcode without injecting any code. In summary, we define a return-into-libc attack as follows.

Return-Into-Libc: *Code-reuse attacks that are based on the principle of return-into-libc subvert the intended control-flow of a program and redirect it to security-critical functions that reside in shared libraries or the executable itself.*

Figure 2.5 shows the typical memory layout of a return-into-libc attack. A necessary step of our specific return-into-libc attack is the injection of the string `/bin/sh` since *system()* is expecting a pointer to the program path in order to open a new terminal. To tackle this issue, one could search for the string inside the

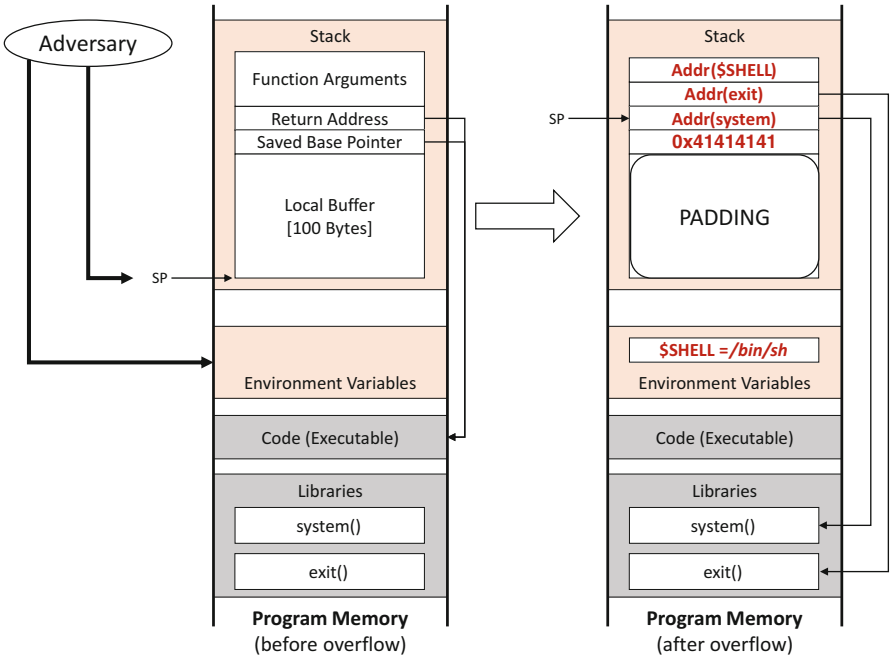

Fig. 2.5 Basic principle of return-into-libc attacks

entire address space of the target application. However, this approach is unreliable as the string might not always be present in the address space of the application. A more robust exploitation approach is to inject the string into a data memory page and record its address. At first glance, this might seem a trivial step. However, the challenge stems from the fact that the string needs to be NULL-terminated. Hence, if we attempt to inject the string into the local buffer, we would need to process a NULL Byte which is impossible for many vulnerabilities. For instance, classic buffer overflow vulnerabilities introduced via the *strcpy()* function terminate the write operation if a NULL Byte is processed. The classic return-into-libc attacks overcome this issue by injecting the string as an environment variable. In Fig. 2.5, the adversary defines the `$SHELL` environment variable which contains the string `/bin/sh`.

After the environment variable has been defined, the adversary interfaces to the application by providing a data input that exceeds the local buffer's limits. Specifically, the adversary fills the local buffer with arbitrary pattern bytes. In addition, the saved base pointer is overwritten with 4 Bytes of arbitrary data. Finally, the return address is replaced with the runtime address of the *system()* function. Moreover, two other addresses are written on the stack: the runtime address of the `libc` *exit()* function, and the runtime address of the `$SHELL` variable. The latter resembles the function argument on the stack frame of the invoked *system()* function. Considering the standard x86-based stack frame layout (see Fig. 2.3), the former will be used as the return address of *system()*. In particular, the *exit()* function will terminate the process upon return of *system()*, i.e., at the time the adversary closes the terminal.

This basic return-into-libc attack requires the knowledge of three runtime addresses. In case no code and data randomization is applied, these addresses can be retrieved by reverse-engineering the application using a debugger. Otherwise, an adversary would need to disclose these addresses using memory disclosure techniques which we discuss in detail in Chap. 4.

The basic return-into-libc attacks only allow invocation of two library library functions, while the second function (in Fig. 2.5 the *exit()* function) needs to be called without any argument. As this poses restrictions on the operations an adversary can perform, several advanced return-into-libc attack techniques have been proposed. For instance, Nergal demonstrated two techniques, called esp-lifting and frame faking, allowing an adversary to perform chained function calls in a return-into-libc attack [30].

2.5.2 Return-Oriented Programming

The above described return-into-libc attack technique has some limitations compared to classic code injection attacks. First, an adversary is dependent on critical `libc` functions such as *system()*, *exec()*, or *open()*. Hence, if we either instrument or eliminate these functions, an adversary would no longer be able to perform a

reasonable attack. In fact, one of the first proposed defenses against return-into-libc is based on the idea of mapping shared libraries to memory addresses that always contain a NULL byte [33]. Second, return-into-libc only allows calling one function after each other. Hence, an adversary is not able to perform arbitrary malicious computation. In particular, it is not possible to perform unconditional branching.

There is also a challenge when applying return-into-libc attacks to 64 Bit based systems (x86-64). On x86-64, function arguments are passed to a subroutine via processor registers rather than on the stack. To tackle this challenge, Krahmer [23] suggested an advanced return-into-libc attack technique called borrowed code chunks exploitation. The main idea is to borrow a function epilogue consisting of several `POP` register instructions. These instructions load the necessary function arguments into processor registers and subsequently redirect the execution-flow to the target subroutine.

Shacham generalizes the idea of borrowed code chunks exploitation by introducing return-oriented programming. This attack technique tackles the previously mentioned limitations of return-into-libc attacks. The basic idea is to execute a chain of short code sequences rather than entire functions. Multiple code sequences are combined to a so-called gadget that performs a specific atomic task, e.g., a load, add, or branch operation. Given a sufficiently large code base, an adversary will most likely identify a gadget set that forms a new Turing-complete language. That said, the derived gadget set can be exploited to induce arbitrary malicious program behavior. The applicability of return-oriented programming has been shown on many platforms including x86 [32], SPARC [4], Atmel AVR [14], PowerPC [25], ARM [22], and z80 [5].

The basic idea and workflow of a return-oriented programming attack is shown in Fig. 2.6. Note that we discuss a return-oriented programming attack that exploits a heap-based vulnerability to explain all basic attack steps that are taken in modern real-world code-reuse exploits. First, the adversary writes the return-oriented payload into the application's memory space, where the payload mainly consists of a number of pointers (the return addresses) and any other data that is needed for running the attack (Step ①). In particular, the payload is placed into a memory area that can be controlled by the adversary, i.e., the area is writable and the adversary knows its start address. The next step is to exploit a vulnerability of the target program to hijack the intended execution-flow (Step ②). In the example shown in Fig. 2.6, the adversary exploits a heap vulnerability by overwriting the address of a function pointer with an address that points to a so-called *stack pivot* sequence [39]. Once the overwritten function pointer is used by the application, the execution-flow is redirected to a stack pivot sequence (Step ③).

Loosely speaking, stack pivot sequences change the value of the stack pointer (`%esp`) to a value stored in another register. Hence, by controlling that register,[1] the

[1]To control the register, the adversary can either use a buffer overflow exploit that overwrites memory areas that are used to load the target register, or invoke a sequence that initializes the target register and then directly calls the stack pivot.

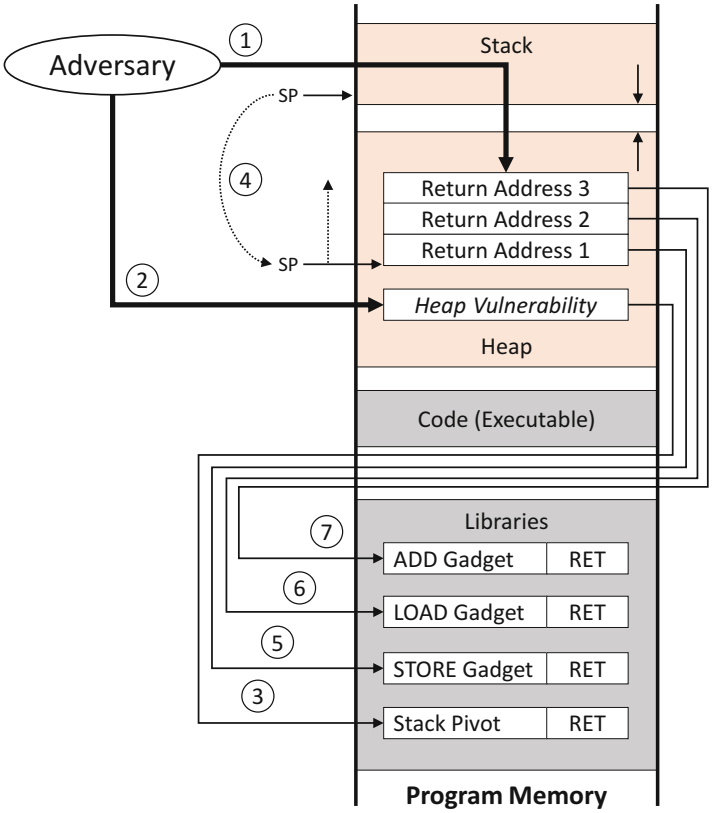

Fig. 2.6 Basic principle of return-oriented programming attacks. For simplicity, we highlight a return-oriented programming attack on the heap using a sequence of single-instruction gadgets

adversary can arbitrarily change the stack pointer. Typically, the stack pivot directs the stack pointer to the beginning of the payload (Step ④). A concrete example of a stack pivot sequence is the x86 assembler code sequence `MOV %esp,%eax; ret`. The sequence changes the value of the stack pointer to the value stored in register `%eax` and subsequently invokes a return (`RET`) instruction. Notice that the stack pivot sequence ends in a `RET` instruction: the x86 `RET` instruction simply loads the address pointed to by `%esp` into the instruction pointer and increments `%esp` by one word. Hence, the execution continues at the first gadget (`STORE`) pointed to by Return Address 1 (Step ⑤). In addition, the stack pointer is increased and now points to Return Address 2.

It is exactly the terminating `RET` instruction that enables the chained execution of gadgets by loading the address the stack pointer points to (Return Address 2) in the instruction pointer and updating the stack pointer so that it points to the next address in the payload (Return Address 3). Steps ⑤ to ⑦ are repeated until the

adversary reaches her goal. To summarize, the combination of different gadgets allows an adversary to induce arbitrary program behavior.

Hence, we define a return-oriented programming attack as follows.

Return-Oriented Programming: *Code-reuse attacks that are based on the principle of return-oriented programming combine and execute a chain of short instruction sequences that are scattered throughout the address space of an application. Each sequence ends with an indirect branch instruction—traditionally, a return instruction—to transfer control from one sequence to the subsequent sequence. In particular, return-oriented programming has been shown to be Turing-complete, i.e., the instruction sequences it leverages can be combined to gadgets that form a Turing-complete language.*

Unintended Instruction Sequences A crucial feature of return-oriented programming on x86 is the invocation of the so-called *unintended* instruction sequences. These can be issued by jumping into the middle of a valid instruction resulting in a new instruction sequence neither intended by the programmer nor the compiler. Such sequences can be found in large number on the x86 architecture due to unaligned memory access and variable-length instructions. As an example, consider the following x86 code with the given intended instruction sequence, where the byte values are listed on the left side and the corresponding assembly code on the right side:

Listing 2.1 Intended code sequence

```
b8 13 00 00 00    MOV $0x13,%eax
e9 c3 f8 ff ff    JMP 3aae9
```

If the interpretation of the byte stream starts two bytes later the following unintended instruction sequence would be executed:

Listing 2.2 Unintended code sequence

```
00 00    ADD %al, (%eax)
00 e9    ADD %ch,%cl
c3       RET
```

In the intended instruction sequence the c3 byte is part of the second instruction. However, if the interpretation starts two bytes later, the c3 byte will be interpreted as a return instruction.

2.6 Hybrid Exploits

Typically, modern systems enforce $W \oplus X$ by default. This forces an adversary to deploy code-reuse attacks. However, a deeper investigation of real-world code-reuse attacks quickly reveals that most exploits today use a combination of code-reuse and

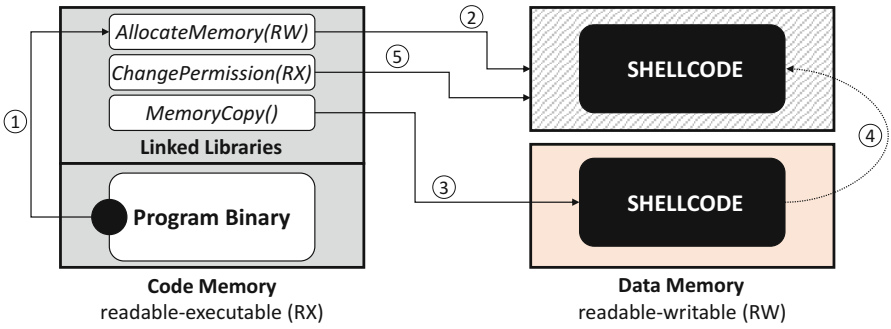

Fig. 2.7 Hybrid exploitation: combination of code-reuse with code injection

code injection. The main idea behind these hybrid exploits is to only use code-reuse attack techniques to undermine $W \oplus X$ protection and launch a code injection attack subsequently. This is possible due to the fact that $W \oplus X$ in its basic instantiation only enforces that a memory page is not writable and executable at the same time. However, a memory page can be first writable (not-executable) and at a later time executable (not-writable).

Figure 2.7 demonstrates this combined attack technique by example. The shown memory layout is divided into a code and data memory area, where the former one is readable and executable, and the latter one is marked as readable and writable. In particular, the code memory holds the program binary and linked shared libraries. In modern operating systems, several important libraries and their functionality are linked by default into the address space of the application. Consider as an example the UNIX C library `libc`. Although the target application may only require the *printf()* function to print strings on the standard output (stdout), other `libc` functions such as *system()* or *memcpy()* will be always mapped into the address space of the application as well.

In our example, we assume that the return-oriented payload and the malicious code have been already injected into the data memory area. In Step ①, the payload exploits a program vulnerability and redirects execution to the shared library segment. Specifically, the adversary invokes a default function to allocate a new memory page (e.g., the *alloc()* function) marked as readable and writable (Step ②). Typically, the return value will be the runtime address of the newly allocated page. Upon return of the memory allocator, the return-oriented payload invokes a memory copy function (e.g., the *memcpy()* function) to copy the injected shellcode to the newly allocated memory page (Step ③ and ④). Finally, the payload invokes a system function (e.g., the *mprotect()* function) to change the memory page permissions of the newly allocated page to readable and executable (Step ⑤). Hence, the adversary can now execute the injected shellcode to perform the actual malicious program actions.

This attack can be further optimized. For instance, if the underlying operating system allows the allocation of read–write-execute (RWX) memory pages to support code generation just-in-time, an adversary can skip the *ChangePermission()* function. Further, it is possible to skip the *AllocateMemory()* and *CopyMemory()* function if the adversary knows the address of the memory page where the shellcode has been originally injected to. In that case, we can simply call the *ChangePermission()* function to mark the corresponding memory page as executable.

2.7 Advanced Code-Reuse Attack Research

Subsequent work demonstrated that Harvard-based architectures—where code and data are strictly separated from each other—cannot prevent return-oriented programming attacks. To this end, Francillon and Castelluccia [14] leverage return-oriented programming to inject arbitrary malware on an Atmel AVR-powered sensor. Further, Buchanan et al. [4] apply return-oriented programming to the RISC-based architecture SPARC, where no unintended code sequences exist by design. In particular, they introduce a compiler that automatically constructs return-oriented exploits. In a similar vein, return-oriented programming has been shown on other architectures including PowerPC-based Cisco routers [25] and ARM-based mobile devices [20, 22]. As real-world example, Checkoway et al. [5] even demonstrate a return-oriented programming exploit on z80-powered voting machines (Harvard architecture) to shift votes.

Hund et al. [19] go one step further: they present the first compiler that automatically identifies return-oriented gadgets in a given binary and compiles (based on the gadget set) return-oriented programs. In particular, they construct a kernel rootkit that entirely leverages return-oriented programming to undermine kernel integrity protection mechanisms. Interestingly, the evaluation of the return-oriented compiler reveals that quicksort executes more than 100 times slower when entirely implemented as return-oriented program. Unfortunately, none of the countermeasures proposed to date has further investigated the tremendous performance overhead of return-oriented programming which could potentially be exploited to detect return-oriented programming execution.

On the one hand, return-oriented programming raised a lot of academic and industrial research. On the other hand, no real-world exploits using return-oriented programming have been discovered until 2010. We believe that this is due to the fact that many PC platforms still did not strictly enforce DEP thereby allowing attackers to launch conventional code injection attacks. However, in 2010, the first return-oriented exploit targeting Adobe PDF has been discovered [21]. From there on, a number of return-oriented exploits have appeared [9, 16, 28, 38].

More distantly related to return-oriented programming is the concept of JIT-spraying attacks [1]. These attacks allow an adversary to return to code she injected via a script. This is achieved by forcing a JIT-compiler to allocate new executable memory pages with attacker-defined code that encapsulates dangerous unintended

instruction sequences. Since scripting languages do not permit an adversary to directly program x86 shellcode, the attacker must carefully construct the script so that it contains useful gadgets in the form of unintended instruction sequences. For instance, Blazakis [1] suggests using XOR operations where the immediate operand to the XOR instruction embeds the malicious instructions. In a recent work, Wilson et al. [24] demonstrate that JIT-spraying attacks are also applicable to architectures that are based on an ARM processor.

2.7.1 Jump-Oriented Programming

All conventional return-oriented programming attacks discussed so far are based on return instructions and thus can be defeated by return address checkers. These tools or compiler extensions ensure the integrity of return addresses, which are corrupted through the conventional return-oriented programming attack [10–12, 15, 17]. However, Checkoway et al. [6] propose a new code-reuse attack that does not require any return instruction. Instead, the attack exploits indirect jump and call instructions on x86 and ARM-based platforms.

Similarly, Bletsch et al. [2] introduce jump-oriented programming (on x86), a code-reuse attack that also requires no return instructions. Bletsch et al. [2] leverage a generic dispatcher gadget to transfer control to the subsequent code sequence. Chen et al. [7] leverage jump-oriented programming to construct a rootkit that does not execute any return instruction.

2.7.2 Gadget Compilers

Gadget compilers ease the adversary's job of identifying gadgets in a given binary, and constructing a return-oriented exploit thereof. We already mentioned two of these gadget compilers: Buchanan et al. [4] introduce a return-oriented exploit compiler for SPARC, and Hund et al. [19] a gadget compiler that automatically identifies gadgets and compiles a return-oriented exploit for x86. However, these two gadget compilers focus on code sequences ending in a return instruction. A gadget compiler that entirely focuses on constructing jump-oriented exploits is presented by Chen et al. [8]. The compiler targets x86-compiled code and leverages the so-called combinational gadget terminating in a `CALL-JMP` sequence to invoke a system call in a jump-oriented attack. Whereas the previously discussed gadget compilers focused on a particular processor platform, Dullien et al. [13] introduce a gadget discovery tool that operates platform-independent by decompiling assembler instructions to an intermediate language called REIL.

Lastly, Schwartz et al. [31] present the Q compiler. This compiler automates the entire process of identifying gadgets, assembling a return-oriented exploit, and hardening existing exploits that fail due to code randomization or DEP. Interestingly,

the Q compiler is based on semantic definitions, e.g., it considers $reg_1 \leftarrow reg_2 * 1$ as a data movement gadget rather than a multiplication gadget allowing Q to compensate missing gadget types. This technique allows Q to generate return-oriented exploits on a small code base (e.g., 20 KB code) that does not per-se contain all gadget types. Related to the small code base leveraged by Schwartz et al. [31], Homescu et al. [18] demonstrate that a Turing-complete gadget set can be derived from so-called microgadgets, i.e., code sequences that only consist of 2–3 Bytes. The probability of finding these very short sequences among different binaries is higher than compared to complex gadgets.

2.7.3 Code-Reuse in Malware

Code-reuse attack techniques have been also leveraged to hide malicious program behavior from static analysis tools or non-ASCII filters. Lu et al. [26] leverage a return-oriented decoder that only consists of code pointers that represent valid ASCII printable characters. The decoder takes as an input the encoded code-reuse exploit and outputs the actual code-reuse exploit at application runtime. Similarly, Wang et al. [37] successfully deployed code-reuse attack techniques to undermine the application vetting process conducted by Apple. To this end, they developed an application that contains an intended buffer overflow vulnerability which gets exploited under certain conditions (e.g., after the app is installed on the user's device). Once the control-flow is hijacked, the exploit payload leverages return-oriented programming to invoke private iOS APIs. Lastly, Vogl et al. [36] introduce persistent data-only malware, i.e., malware that leverages code-reuse attack techniques to realize a persistent rootkit. With respect to malware detection, Stancill et al. [35] present an analysis system that detects return-oriented programming payloads in malicious files. Their system efficiently analyzes incoming documents (PDF, Office, or HTML files), and detects whether they contain a return-oriented programming payload.

In a different domain, code-reuse techniques have been deployed to hide secret program actions: Lu et al. [27] introduce program steganography that is based on executing unintended code sequences to hide program actions from static analysis tools. However, leveraging code-reuse attack techniques for a legitimate and benign purpose will eventually lead to false alarms in tools that aim at detecting and preventing code-reuse attacks.

This concludes our discussion on research on code-reuse attacks. However, note that we will discuss more advanced code-reuse attacks in the subsequent chapters. In particular, we will elaborate on those attacks that bypass defenses based on control-flow integrity enforcement or fine-grained code randomization.

References

1. Blazakis, D.: Interpreter exploitation. In: Proceedings of the 4th USENIX Conference on Offensive Technologies, WOOT'10 (2010). http://dl.acm.org/citation.cfm?id=1925004.1925011
2. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS'11 (2011). http://doi.acm.org/10.1145/1966913.1966919
3. Brunette, G.: Solaris non-executable stack overview. https://blogs.oracle.com/gbrunett/entry/solaris_non_executable_stack_overview (2007)
4. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to RISC. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS'08 (2008). http://doi.acm.org/10.1145/1455770.1455776
5. Checkoway, S., Feldman, A.J., Kantor, B., Halderman, J.A., Felten, E.W., Shacham, H.: Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In: Proceedings of the 2009 Conference on Electronic Voting Technology/-Workshop on Trustworthy Elections, EVT/WOTE'09 (2009). http://dl.acm.org/citation.cfm?id=1855491.1855497
6. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS'10 (2010). http://doi.acm.org/10.1145/1866307.1866370
7. Chen, P., Xing, X., Mao, B., Xie, L.: Return-oriented rootkit without returns (on the x86). In: Information and Communications Security. Lecture Notes in Computer Science, vol. 6476 (2010). http://link.springer.com/chapter/10.1007%2F978-3-642-17650-0_24
8. Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., Yin, X.: Automatic construction of jump-oriented programming shellcode (on the x86). In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS'11 (2011). http://doi.acm.org/10.1145/1966913.1966918
9. Chen, X., Caselden, D., Scott, M.: The dual use exploit: CVE-2013-3906 used in both targeted attacks and crimeware campaigns. https://www.fireeye.com/blog/threat-research/2013/11/the-dual-use-exploit-cve-2013-3906-used-in-both-targeted-attacks-and-crimeware-campaigns.html (2013)
10. Chiueh, T., Hsu, F.H.: RAD: a compile-time solution to buffer overflow attacks. In: Proceedings of the 21st International Conference on Distributed Computing Systems, ICDCS'01 (2001). http://dl.acm.org/citation.cfm?id=876878.879316
11. Chiueh, T., Prasad, M.: A binary rewriting defense against stack based overflow attacks. In: Proceedings of the 2003 USENIX Annual Technical Conference, ATC'03 (2003). https://www.usenix.org/legacy/event/usenix03/tech/full_papers/prasad/prasad_html/camera.html
12. Davi, L., Sadeghi, A.R., Winandy, M.: ROPdefender: a detection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS'11 (2011). http://doi.acm.org/10.1145/1966913.1966920
13. Dullien, T., Kornau, T., Weinmann, R.P.: A framework for automated architecture-independent gadget search. In: Proceedings of the 4th USENIX Conference on Offensive Technologies, WOOT'10 (2010). http://dl.acm.org/citation.cfm?id=1925004.1925012
14. Francillon, A., Castelluccia, C.: Code injection attacks on Harvard-architecture devices. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS'08 (2008). http://doi.acm.org/10.1145/1455770.1455775
15. Frantzen, M., Shuey, M.: StackGhost: hardware facilitated stack protection. In: Proceedings of the 10th USENIX Security Symposium (2001). http://dl.acm.org/citation.cfm?id=1251327.1251332

16. Goodin, D.: Apple quicktime backdoor creates code-execution peril. http://www.theregister.co.uk/2010/08/30/apple_quicktime_critical_vuln/ (2010)
17. Gupta, S., Pratap, P., Saran, H., Arun-Kumar, S.: Dynamic code instrumentation to detect and recover from return address corruption. In: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, WODA'06, pp. 65–72 (2006). http://doi.acm.org/10.1145/1138912.1138926
18. Homescu, A., Stewart, M., Larsen, P., Brunthaler, S., Franz, M.: Microgadgets: size does matter in Turing-complete return-oriented programming. In: Proceedings of the 6th USENIX Conference on Offensive Technologies, WOOT'12 (2012). http://dl.acm.org/citation.cfm?id=2372399.2372409
19. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: Proceedings of the 18th Conference on USENIX Security Symposium (2009). http://dl.acm.org/citation.cfm?id=1855768.1855792
20. Iozzo, V., Miller, C.: Fun and games with Mac OS X and iPhone payloads. In: Black Hat Europe (2009). http://www.blackhat.com/presentations/bh-europe-09/Miller_Iozzo/BlackHat-Europe-2009-Miller-Iozzo-OSX-IPhone-Payloads-whitepaper.pdf
21. jduck: The latest Adobe exploit and session upgrading. http://bugix-security.blogspot.de/2010/03/adobe-pdf-libtiff-working-exploitcve.html (2010)
22. Kornau, T.: Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University Bochum (2009). http://static.googleusercontent.com/media/www.zynamics.com/en//downloads/kornau-tim--diplomarbeit--rop.pdf
23. Krahmer, S.: x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://users.suse.com/~krahmer/no-nx.pdf (2005)
24. Lian, W., Shacham, H., Savage, S.: Too lejit to quit: extending JIT spraying to ARM. In: 22nd Annual Network and Distributed System Security Symposium, NDSS'15 (2015). http://www.internetsociety.org/doc/too-lejit-quit-extending-jit-spraying-arm
25. Lindner, F.: Router exploitation. http://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-SLIDES.pdf (2009)
26. Lu, K., Zou, D., Wen, W., Gao, D.: Packed, printable, and polymorphic return-oriented programming. In: Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection, RAID'11 (2011). http://dx.doi.org/10.1007/978-3-642-23644-0_6
27. Lu, K., Xiong, S., Gao, D.: Ropsteg: program steganography with return oriented programming. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, CODASPY'14 (2014). http://doi.acm.org/10.1145/2557547.2557572
28. Marschalek, M.: Dig deeper into the ie vulnerability (cve-2014-1776) exploit. https://www.cyphort.com/dig-deeper-ie-vulnerability-cve-2014-1776-exploit/ (2014)
29. Microsoft: Data execution prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/ (2006)
30. Nergal: The advanced return-into-lib(c) exploits: PaX case study. Phrack Mag. **58**(4) (2001). http://www.phrack.org/issues.html?issue=58&id=4#article
31. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: Exploit hardening made easy. In: Proceedings of the 20th USENIX Security Symposium (2011). http://dl.acm.org/citation.cfm?id=2028067.2028092
32. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS'07 (2007). http://doi.acm.org/10.1145/1315245.1315313
33. Solar Designer: lpr LIBC RETURN exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html (1997)
34. Solar Designer: Non-executable stack patch. http://lkml.iu.edu/hypermail/linux/kernel/9706.0/0341.html (1997)
35. Stancill, B., Snow, K., Otterness, N., Monrose, F., Davi, L., Sadeghi, A.R.: Check my profile: leveraging static analysis for fast and accurate detection of rop gadgets. In: Research in Attacks, Intrusions, and Defenses. Lecture Notes in Computer Science, vol. 8145 (2013). http://dx.doi.org/10.1007/978-3-642-41284-4_4

36. Vogl, S., Pfoh, J., Kittel, T., Eckert, C.: Persistent data-only malware: function hooks without code. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS'14 (2014). http://www.internetsociety.org/doc/persistent-data-only-malware-function-hooks-without-code
37. Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on iOS: when benign apps become evil. In: Proceedings of the 22nd USENIX Security Symposium (2013). http://dl.acm.org/citation.cfm?id=2534766.2534814
38. Westin, K.: GnuTLS crypto library vulnerability CVE-2014-3466. http://www.tripwire.com/state-of-security/latest-security-news/gnutls-crypto-library-vulnerability-cve-2014-3466/ (2014)
39. Zovi, D.D.: Practical return-oriented programming. SOURCE Boston. http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf (2010)