CHAPTER

Stairway to Successful Kernel Exploitation

3

INFORMATION IN THIS CHAPTER

- A Look at the Architecture Level
- The Execution Step
- The Triggering Step
- The Information-Gathering Step

INTRODUCTION

In Chapter 2, we said a bug becomes a security issue as soon as someone figures out how to take advantage of it. That's what we'll focus on in this chapter: how to develop a successful exploit. Demonstrating that a vulnerability exists (e.g., via proof-of-concept code) is only a first step in kernel exploitation. The exploit has to *work*. A piece of code that gives you full privileges and then immediately panics the machine is clearly of no use.

To develop a good exploit, you must understand the vulnerability you are targeting, the kernel subsystems involved, and the techniques you are using. A properly written exploit has to be:

- **Reliable** You should narrow down, as much as possible, the list of preconditions which must be met for the exploit to work, and design the code to always generate those preconditions. The fewer variables you depend on, the more likely you will be able to generate the desired situation. Ideally, if some condition is not under your control (or might change from execution to execution), you should know why.
- **Safe** You must identify what part of the exploit might crash the machine, and try to detect that at runtime. The exploit code should be as conservative as possible and defend itself in those scenarios. Also, once executed, it should leave the machine in a stable state.
- **Effective** You should always aim to achieve the most you can from the vulnerability. If the vulnerability can lead to code execution (or any other privilege gain) crashing the machine is not enough. The exploit also should be *portable*, which means it should work on as many targets as possible. This is usually a direct consequence of how small you managed to make the set of variables on which you depend.

A Guide to Kernel Exploitation. DOI: 10.1016/B978-1-59749-486-1.00003-6

Since we already focused on understanding vulnerabilities in Chapter 2, we're ready now to dive deep into the realm of exploit development. To summarize what we discussed in Chapter 1, exploit development comprises three main steps: the *preparatory step*, the *trigger step*, and the *execution step*. Each step creates the conditions necessary for the following step to succeed. For this reason, we will work our way backward through the steps, starting our analysis from the execution phase, to clarify *what* a step tries to achieve and *how* proper implementation of the first two steps can increase your chances of success when it comes time to execute the exploit. But before we start, let's discuss another protagonist that influences both the kernel and our attempts at attacking it: the *architecture level*.

By *architecture*, we refer mainly to how the CPU behaves: what instructions it can execute, which instructions are privileged, how it addresses memory, and so on. For our purposes, we will focus mostly on the 64-bit variant of x86 family, the *x86-64* architecture (we'll discuss our reason for focusing on this architecture in the following section). In this chapter (as well as throughout Part I of the book), our goal is to be as operating-system-independent as possible, focusing on the ideas and the theoretical background behind the various approaches used during exploit development, and leaving the dirty implementation details (and issues) to the subsequent, practical, chapters (Chapters 4 through 8). In an environment as complex and dynamic as any modern kernel is, techniques come and go, but building a good methodology (an approach toward exploitation) and understanding the ideas behind specific techniques will allow you to adapt the practical techniques described in the subsequent chapters to different scenarios or future kernel versions.

A LOOK AT THE ARCHITECTURE LEVEL

No serious exploit development analysis can begin without considering the underlying architecture to the kernel you're targeting. This is especially true for kernel-land exploitation, where the target, the kernel, is the piece of software that is closest to the machine. As we noted earlier, *architecture* refers to the operations of the CPU and the hardware memory management unit (MMU). Since this book is about writing exploits more than designing CPUs, we'll focus only on the details that are relevant to our discussion. For more information on computer architecture principles and practical implementation, please see the "Related Reading" section at the end of this chapter.

Generic Concepts

Before getting into the details of our architecture of choice, let's recap the generic concepts that apply to all architectures so that our analysis will be clearer.

CPU and Registers

The CPU's role is extremely simple: execute instructions. All the instructions that a CPU can execute comprise the architecture's *instruction set*. At the very

least, a typical instruction set provides instructions for arithmetic and logic operations (*add*, *sub*, *or*, *and*, etc.), control flow (*jump/branch*, *call*, *int*, etc.), and memory manipulation (*load*, *store*, *push*, *pop*, etc.). Since accessing memory is usually a slow operation (compared to the speed at which the CPU can crank instructions), the CPU has a set of local, fast registers. These registers can be used to store temporary values (*general-purpose registers*) or keep relevant control of information and data structures (*special-purpose registers*). CPU instructions usually operate on registers.

Computer architectures are divided into two major families: *RISC* (Reduced Instruction Set Computer), which focuses on having simple, fixed-size instructions that can execute in a clock cycle; and *CISC* (Complex Instruction Set Computer), which has instructions of different sizes that perform multiple operations and that can execute for more than a single clock cycle. We can further differentiate the two based on how they access memory: RISC architectures require memory access to be performed through either a *load* (copy from memory) or a *store* instruction, whereas CISC architectures may have a single instruction to access memory and, for example, perform some arithmetic operation on its contents. For this reason, RISC architectures are also usually referred to as *load-store architectures.* On RISC architectures, apart from load, store, and some control flow instructions, all the instructions operate solely on registers.

NOTE

Today the distinction between RISC and CISC is blurry, and many of the issues of the past have less impact (e.g., binary size). As an example, all recent x86 processors decode complex instructions into micro-operations (micro-ops), which are then executed by what is pretty much an internal RISC core.

The CPU fetches the instructions to execute from memory, reading a stream of bytes and decoding it accordingly to its instruction set.[A] A special-purpose register, usually called the *instruction pointer* (*IP*) or *program counter* (*PC*), keeps track of what instruction is being executed.

As we discussed in Chapter 2, a system can be equipped with a single CPU, in which case it is referred to as a uniprocessor (UP) system, or with multiple CPUs, in which case it is called a symmetric multiprocessing (SMP) system.[B] SMP systems are intrinsically more complex for an operating system to handle, since

[A]We try to keep the discussion simple here, but it's worth mentioning that the process of fetching, decoding, and executing is divided into independent units and is highly parallelized through the use of pipelines to achieve better performance.

[B]A characteristic of multiprocessor systems is that all of the processors can access all of the memory, either at the same speed (Uniform Memory Access [UMA]) or at different speeds (Non-Uniform Memory Access [NUMA]) depending on the location. Other configurations with multiple CPUs also exist; for example, cluster processors, where each CPU has its own private memory.

now *true* simultaneous execution is in place. From the attacker's point of view, though, SMP systems open more possibilities, especially when it comes to winning race conditions, as we will discuss later in this chapter.

Interrupts and Exceptions

The CPU blindly keeps executing whatever is indicated at the IP/PC, each time incrementing its value by the size of the instruction it has decoded. Sometimes, though, the CPU stops or is interrupted. This occurs if it encounters an error (e.g., an attempt to divide by zero), or if some other component in the system (e.g., a hard drive) needs attention. This interruption can thus be either *software-generated* or *hardware-generated*. All modern architectures provide an instruction to explicitly raise an interrupt. Interrupts generated by an error condition (as in the divide-by-zero case) are called *exceptions*, and interrupts generated by software are generally known as *traps*. Software-generated interrupts are *synchronous*: given a specific path, they will always occur at a specific time, as a consequence of executing a specific instruction. Hardware-generated interrupts are *asynchronous*: they can happen unpredictably, at any time.

Interrupts and exceptions are identified by an integer value. The CPU usually provides a special-purpose register to keep track of the memory address of a table, the *interrupt vector table*, which associates a specific routine (an *interrupt* or *exception handler*) to each interrupt. By registering a routine, the operating system can be notified each time an interrupt occurs and have the flow of execution redirected to the address stored in the table. Thanks to this approach, the system can react to (and handle) specific interrupts.

Modern CPUs have at least two modes of operation: *privileged* and *unprivileged*. In privileged mode, the whole instruction set is available, whereas in unprivileged mode only a subset of it can be used. Kernel code runs in privileged mode. Unprivileged code can request a service to some privileged code by executing a specific interrupt or an instruction provided by the architecture.

Memory Management

Just as the CPU fetches the stream of instructions from memory, it also fetches load/store operations on a RISC machine and many different instructions on a CISC machine. Let's discuss this in more depth and see, from an architecture point of view, how this memory is managed.

Simply put, memory is a sequence of bytes, each of which is assigned a positive numeric incremental number, starting with zero. This number represents the *address* of the specific byte. Instructions accessing memory use the address to read or write at a specific location. For example, the IP/PC register mentioned earlier stores the address of the next location in memory from which the CPU will fetch the next instruction. Such numeric addressing is usually referred to as *physical addressing* and ranges from 0 to the amount of physical memory installed.

The CPU can specify a physical address in two main ways:

- **Linearly** The entire physical range is presented as a single consecutive sequence of bytes. This approach can be as simple as a direct 1:1 mapping between the physical and the linear address ranges, or it can require techniques to generate a virtual address space and translate from one to the other (*paging* is the classic example here, as we will discuss shortly). This is the approach used nearly everywhere today.
- **Segmentation based** The entire physical range is presented as a collection of different segments. To reference a specific physical address the CPU needs to use at least two registers: one holding the segment base address (usually stored in a table so that it can be retrieved by its segment number) and an offset inside that segment. Thanks to this approach, at parity of register size, segmentation allows a lot more memory to be addressed than the linear address model approach does. In the days of 16-bit computing, this was a huge plus. Today, with 32-bit and 64-bit models, this is no longer the case, and in fact, segmentation has almost not been used at all in modern operating systems. The 64-bit version of the x86 architecture has greatly limited segmentation support.

Central to paging are the *page*, a unit of memory, and the use of *page tables*, which describe the mapping between physical addresses and linear addresses. Each linear address is divided into one or more parts, each corresponding to a level in the page tables, as you can see in Figure 3.1. Two or three levels are common on 32-bit architectures, whereas four levels are usually used on 64-bit architectures.

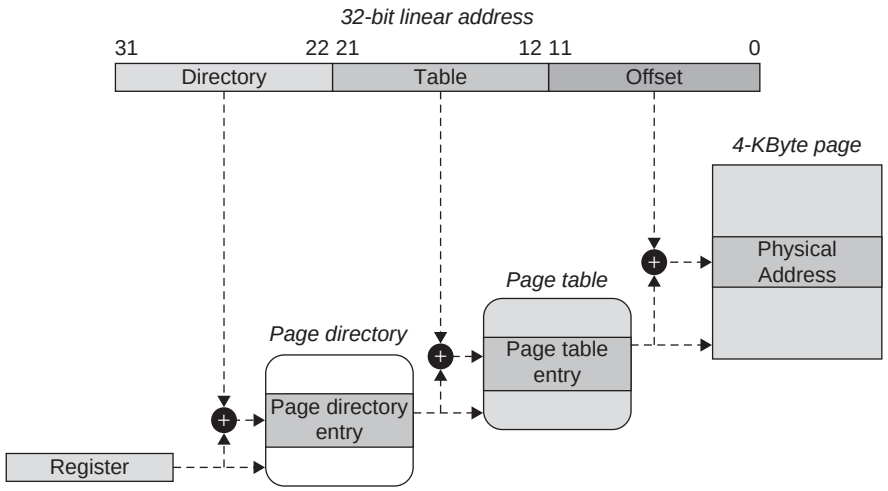

FIGURE 3.1

Two-level paging with 32-bit virtual addresses.

The last part of the virtual address (in Figure 3.1, the last 12 bits) specifies an offset inside the page, and the previous parts of the virtual address (the first 20 bits in Figure 3.1) specify one index (or more, depending on the number of levels) inside the page tables. When a linear address is used inside an instruction, the CPU sends the linear address to the MMU, whose job is to walk the page tables and return the physical address associated with the specific entry. To do that, the MMU needs to identify the set of page tables in use, through the physical address stored inside one of the special-purpose registers. Operating systems exploit this feature to give the illusion of a separate linear address space to each process. The system allocates space for each process's page tables and, at each context switch, copies the physical address of the current process's page tables in the special-purpose register.

Virtual-to-physical address translation is mandatory for a CPU to work correctly; however, it is an expensive operation. To improve the performance of this recurrent operation, architectures offer a cache of the most recent virtual-to-physical associations, called the *translation lookaside buffer* (*TLB*). The idea behind a TLB is pretty simple: keep the result of a page lookup for a specific virtual address so that a future reference will not have to go through the MMU walking mechanism (and will not have to access the physical memory addresses where page tables are stored). As with any cache, TLBs exploit the principle of *locality*, both *temporal* and *spatial*: it is likely that a program will access data around the same address in the near future. As a classic example of this, think of a loop accessing the various members of an array. By caching the physical address of the array there is no need to perform an MMU translation at each member access.

Operating systems create the illusion of a private virtual address space for each process. As a result, the same virtual address will almost always have different translations in different processes. Actually, such virtual addresses may not even exist in some. If the TLB associations were kept between each context switch, the CPU could end up accessing the wrong physical addresses. For that reason, all architectures provide a means to *flush* either the TLB cache or a specific TLB entry. Architectures also provide a way to save a TLB entry across flushes (for virtual-to-physical mappings that do not change across context switches) to enable global entries.

As you can imagine, flushing the TLB creates a performance impact. Returning to the array loop example, imagine two processes going through two long arrays and becoming interleaved. Each time a context switch occurs between the two, the next attempt to access a member of the array requires an MMU walk of the page tables.

From the point of view of the MMU, the operating system accesses memory through its own page tables, just like any user-land process. Since going back and forth from user land to kernel land is an extremely common task, this translates to flushing the TLB cache not only at each process context switch, but also at each entry/exit from kernel land. Moreover, the kernel usually needs user-land access—for example, to bring in the arguments of a call or return the results of a call. On architectures such as the x86/x86-64 that do not provide any hardware support to

access the context of another process, this situation translates into TLB flushes at each kernel entry/exit and the need to manually walk the page tables each time a reference to another context is needed, with all the associated performance impacts.

To improve performance on such architectures (which is always a key point in operating system design), operating systems implement the combined user/kernel address space mentioned in Chapter 1 and replicate kernel page tables on top of each process. These page translations (from kernel virtual addresses to physical ones) are then marked as global in the TLB and never change. They are simply protected by marking them as accessible from privileged code only. Each time a process traps to kernel land there is no need to change the page tables (and thus flush the TLB cache); if for some reason the kernel directly dereferences a virtual address in the process context and this address is mapped, it will just access the process memory.

Some architectures (e.g., SPARC V9) instead provide support for accessing a context from inside another context and to associate TLB entries to specific contexts. As a result, it is possible to separate user land and kernel land without incurring a performance impact. We will discuss the implications of these designs in the section "The Execution Step."

WARNING

Although a combined user/kernel-land design is the common choice on x86, this choice is driven primarily for performance reasons: implementing proper separation between kernel land and user land is entirely possible. The 4G/4G split project for the Linux Kernel, the PaX project, and, even more interestingly, the Mac OS X operating system are examples of implementations of separate user-land and kernel address space on the x86 architecture. The x86-64 architecture has changed the landscape a bit. With a lot of virtual address space available, there is plenty of space for both kernel land and user land, and the limited support for segmentation has made it impossible to use segmentation-based tricks to achieve good performance in a separate environment (as PaX does on x86).

The Stack

The *stack* is a memory structure that is at the base of nearly any *Application Binary Interface* (*ABI*), the set of rules that mandate how executables are built (data type and size, stack alignment, language-specific constructs, etc.) and behave (calling convention, system call number and invocation mechanisms, etc.). Since the kernel is an executable itself, we will cover the parts of the ABI that affect our exploitation approaches the most, focusing in particular on the *calling convention*.

The calling convention specifies how the glue mechanism that is necessary to support nested procedures is put together; for example, how parameters and return values are passed down or how control is transferred back to the caller correctly when a procedure exits. All the architectures vary slightly regarding how they support implementing nested procedures, but a common component is the stack.

The stack is based on two operations:

- ***PUSH*** Places a value at the top of the stack
- ***POP*** Removes the value at the top of the stack and returns it to the caller

Due to this design, the stack behaves as a *LIFO* (*last in, first out*) data structure. The last object we *PUSH* on the stack is the one that we get back at the next *POP* operation. Traditionally, the stack grows from higher addresses toward lower addresses, as you saw in Chapter 2. In such a case, the *PUSH* operation subtracts the object size from the *TOS* (*top of the stack*) and then copies the object at the pointed address, while the *POP* operation reads the value pointed to by the *TOS* and then increments its value with the object size.

Architectures have a register dedicated to holding the *TOS* value and provide *POP* and *PUSH* instructions that implicitly manipulate the *TOS* register. Figure 3.2 shows how these architectural features can be used to support nested procedures.

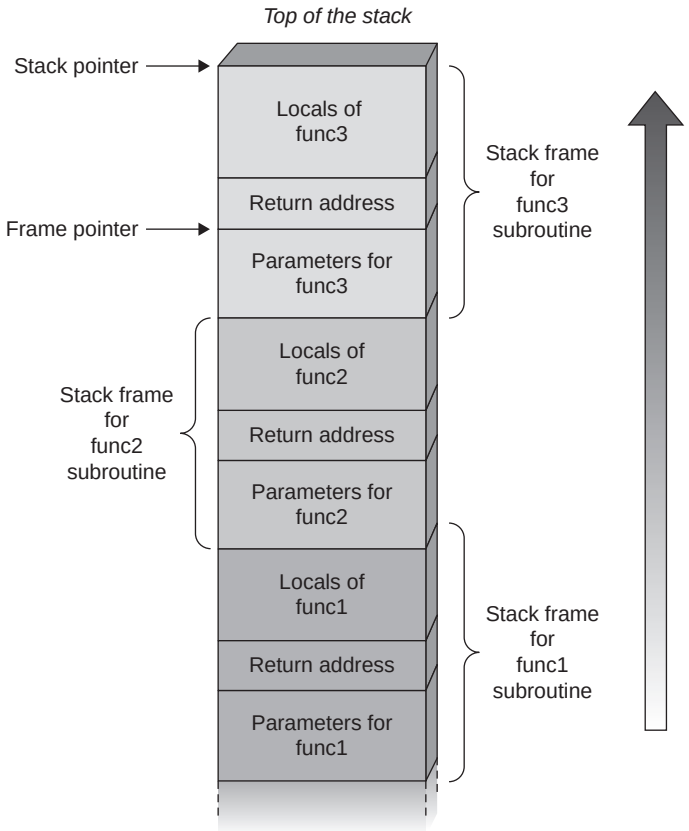

FIGURE 3.2

Nested procedures implemented through a stack.

The idea is to confine each procedure into a s*tack frame*, a portion of the stack that is private to the procedure. This private area can be used to store local variables by simply reserving enough space to hold them within the stack frame. Right before calling a procedure, the caller places the IP of the next instruction after the call on the stack. Once the callee (the called function) terminates, it cleans the stack that it has been locally using and pops the next value stored on top of the stack. This value is the address of the next instruction in the caller that the caller itself pushed previously. The callee sets the IP to this value and the execution continues correctly.

Although passing parameters to functions is commonly done via registers, especially on RISC architectures that have many registers, on some architectures, such as the x86 32-bit architecture, the stack can also be used to do that. The caller simply pushes the parameters on the stack and then the callee pops them back. This use of the stack is the one presented in Figure 3.2. In this case, the callee cleans the stack by removing the parameters. Since the stack is simply a memory structure, the callee can also access the parameters via an offset from the top of the stack without popping them out. In this case, it is up to the caller to clean the stack once the callee returns. The former approach is typical on x86 Windows systems, whereas the latter approach is more common on x86 UNIX systems.

x86 and x86-64

Now that we've recapped generic architecture concepts, it is time to see how our architectures of choice implement them. This discussion will lead the way to the first step we will cover in exploit development, the execution step.

The 32-bit x86 Architecture

The most famous CISC architecture is also the one you probably are most familiar with: x86. The first example of this architecture dates back to 1978, when the Intel 8086 16-bit processor was released.[C] This link still lingers today in modern x86 CPUs. When you switch on your computer, the CPU boots in Real Mode, a 16-bit environment that is pretty much the same as the 8086 one. Backward compatibility has always been mandatory in x86 design and it is the reason for both its success and its awkwardness. Customers are very happy to be able to keep running their old legacy applications, and they couldn't care less about the current state of the instruction set.

On x86, one of the first things your system does after it starts executing is to switch to Protected Mode, the 32-bit environment your operating system is running in. From an operating system point of view, Protected Mode is a godsend, providing such features as a paging MMU, privilege levels, and a 32-bit addressable virtual address space. In 32-bit Protected Mode, the x86 offers eight 32-bit

[C] http://download.intel.com/museum/archives/brochures/pdfs/35yrs_web.pdf

general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP), six 16-bit segment registers (CS, DS, ES, FS, GS, and SS), and a variety of special-purpose registers. The registers you will likely have to deal with are:

- **ESP/EBP** These hold the stack pointer (ESP) and the frame pointer (EBP). The first one points to the top of the current stack, while the second one points to the "entry point" of the current function. The EBP is then used to reference the parameters passed to the function and the local variables. It is worth mentioning that using the EBP as a frame pointer is not mandatory; in fact, kernels generally get compiled without using the frame pointer, to have an extra temporary register.
- **EIP** This holds the instruction pointer.
- **EFLAGS** This keeps bit flags mostly relative to the current execution state.
- **CR0–CR7** These are control registers, which hold configuration bits for the running system. *CR3* holds the physical address of the current page tables.
- **IDTR** This is the interrupt descriptor table register, which holds the physical address of the interrupt descriptor table (IDT), the table that associates a service routine to each interrupt. The *lidt* (unprivileged) and *sidt* (privileged) instructions allow writing and reading from the IDTR.
- **GDTR** This is the global descriptor table register, which holds the physical address of the global descriptor table (GDT), which is a table of segment descriptors. Because of how x86 is designed, the GDT is mandatory (and thus will always be present in any operating system). *sgdt* and *lgdt* behave with the GDT just like *sidt* and *lidt* do with the IDT.

The x86 architecture has four privilege levels, called *rings*. Ring 0 is the most privileged level and it is the one the kernel runs in. User-land programs run at Ring 3, the least privileged of the levels. Rings 1 and 2 are rarely used by modern operating systems.

The x86 architecture supports both *paging* and *segmentation*. Actually, segmentation cannot be disabled in Protected Mode, so addresses on x86 are always of the form *seg:offset*, where *seg* is one of the six segment registers. Anytime a segment register is not specified, an *implicit* segment register is used: *CS* is the implicit segment register for instruction fetching, *DS* is the one for data access, *SS* is the one for stack manipulation, and *ES* is the one for string instructions. To have a single linear address space, operating systems have all the segments defined with base address 0 and segment limit 0xFFFFFFFF, thereby creating a single large segment that spans the entire 4GB virtual address space. Paging is then used to efficiently implement virtual memory on top of it.

The x86 architecture implements two-level page tables (three if Physical Address Extension (PAE) is enabled, although we won't go into the details here). The *CR3* register holds the physical address of the page directory table (PDT) in use. The first 10 most significant bits of a linear address are used as an index inside the PDT, to pick one of the 1,024 (2^{10}) entries. Each entry holds the physical address of a page table (PT). The next 10 most significant bits of a linear

address space select an entry in the PT. This entry is usually called the page table entry (PTE) and contains the physical address of the searched page. The remaining 12 bits act as an offset inside the physical page, to address each of the 4,096 bytes that compose the page. The MMU performs this operation automatically each time it gets a linear address from the CPU.

Associated with each PTE are a bunch of flags that describe the page. The most interesting of these flags are the ones specifying page protections. On the x86 architecture, a page can be *READABLE* and/or *WRITABLE*; there is no support to mark whether a page is *EXECUTABLE* (all accessible pages are implicitly *EXECUTABLE*). As you will see in this chapter, this is an interesting property.

Also interesting to note is that the x86 architecture provides a general flag, known as *WP* (*Write Protect*), inside *CR0* that, when set, prevents privileged code from modifying any read-only page, regardless of whether it is in a privileged or an unprivileged segment. This flag is turned on by default on all modern kernels.

x86-64

As applications began to demand larger address spaces and RAM prices began to drop, Intel and AMD started to pursue 64-bit architectures. Intel developed the brand-new *IA64* RISC architecture; AMD took the x86 32-bit architecture, put it on 64-bit steroids (64-bit registers and integer operations, a 64-bit address space, etc.), and called it *AMD64*. AMD64 is completely backward-compatible, allowing users to run 32-bit applications and operating systems unmodified, and has two main modes of operation:

- **Legacy Mode** The CPU behaves like a 32-bit CPU and all the 64-bit enhancements are turned off.
- **Long Mode** This is the native 64-bit mode of operation. In this mode, 32-bit applications can still run unmodified (discussed shortly), in a mode referred to as *Compatibility Mode*. In Compatibility Mode, it is easy (and fast enough) to switch to the full 64-bit mode and back. The Mac OS X kernel (up to Snow Leopard) has used this feature to run 64-bit applications and (mainly) a 32-bit kernel.

Not entirely surprisingly, AMD64 was so much more successful than IA64 that Intel had to develop its own compatible version of it, known as *EM64T/IA-32e*. The differences between the two were minimal, and we will not cover them here. Today, the 64-bit version of the 32-bit architecture is generally referred to as *x86-64*.

Now let's discuss those aforementioned 64-bit steroids:

- The 32-bit general-purpose registers (EAX, EBX, etc.) have been extended to 64-bit and are called RAX, RBX, and so on.
- Eight new 64-bit registers have been added, named R8 to R15.
- A *nonexecute* (*NX*) bit is present by default to mark pages as nonexecutable. The NX bit was already available on some x86 32-bit processors when PAE was enabled.

- It is now possible to use the RIP (64-bits version of the EIP register) to reference memory relative to the instruction pointer. This is an interesting feature for *position-independent code* (code that does not make any absolute address reference and can thus be placed anywhere in the address space and be executed correctly).
- The virtual address space is obviously larger. Since a 64-bit address space might put a bit too much pressure on the memory structures used to represent it (e.g., page tables), a subset of it is used; namely, "only" 2^{48} addresses are used. This is achieved by having the remaining 16 bits set as a copy of the 47th bit, thereby generating a virtual memory hole between 0x7FFFFFFFFFFF and 0xFFFF800000000000. Operating systems commonly use this to separate user land and kernel land, giving the lower portion to the user and the upper portion to the kernel.
- Page table entries are now 64 bits wide (as happens on x86 when PAE is enabled), so each level of indirection holds 512 entries. Pages can be 4,096KB, 2MB, or 1GB in size. A new level of indirection is necessary, called *PML4*.
- In 64-bit Long Mode, segmentation has been largely crippled. As an example, the GDT remains, but a lot of the information stored in it (e.g., segment limit and access type) is simply ignored. The GS and FS segment selector registers also remain, but they are generally used only to save/store an offset to important data structures. In particular, GS is generally used both in user land and kernel land because the architecture offers an easy way to switch its value upon entering/exiting the kernel: *SWAPGS*. We will discuss the use of SWAPGS in more detail in Part II of the book.
- The calling convention procedure has changed. Whereas on the x86 architecture parameters are generally passed on the stack (unless the compiler decides differently for some functions, generally leaf functions, as a consequence of some specified optimization), the x86-64 ABI dictates that the majority of parameters get passed on registers. We will come back to this topic when we talk about stack exploitation later in this chapter.

It is also important to remember that, apart from the differences we mentioned earlier, nearly everything we have discussed regarding the x86 architecture holds true on x86-64 as well.

THE EXECUTION STEP

Now that we've discussed the architecture, it's time to discuss the execution step. As noted earlier, in many exploits this step can be further divided into two substeps:

- **Gaining privileges** This means raising the privileges (or obtaining more privileges) once they are executed. As we will discuss later in this section, the most common operation in kernel land is to locate the structures that keep

track of the process credentials and raise them to super-user credentials. Since the code is executing at kernel land with full privileges, all the user-land (and nearly all the kernel-land) protections can be circumvented or disabled.

- **Fixating the system** This means leaving the system in a stable state so that the attacker can enjoy his or her freshly gained privileges. As we will discuss shortly, execution of privilege-gaining code is generally a consequence of a redirection of execution flow. In other words, you may end up leaving a kernel path before it has completed. If this is the case, whatever resource the kernel path grabbed (especially locks) may need to be properly restored. The more an exploit disrupts the kernel state, the more emulation/fixating code needs to be written to keep the system up and running correctly. Moreover, with memory corruption bugs, it may take some "time" from when you perform the overflow to when your hijacking of the control flow takes place. If any of the memory that you overwrote is accessed in between and checked against some value, you must make those checks pass.

As we stated in Chapter 1, shellcode is just a handful of assembly instructions to which you want to redirect execution flow. Obviously, though, you need to place these instructions in memory and know their address so that you can safely redirect the flow there. If you make a mistake in picking up the destination address, you will lose the target machine.

Placing the Shellcode

Since losing target machines is not our main objective, let's look at our range of options for safely and reliably placing the shellcode. Depending on both the vulnerability type (the class it belongs to, how much control it leaves) and the memory model in use (either separated or combined user/kernel address space), you may place your shellcode in either the kernel address space or the user address space, or a mix of the two.

As usual, kernel land imposes some constraints that you have to carefully respect:

- *The hijacked kernel path must be able to see the memory location of the shellcode*. In other words, the shellcode must be in the range of virtual address spaces that the kernel can directly access using the current set of page tables. This basically translates to placing the shellcode into the sole kernel context on systems implementing the user/kernel split address space model, and into the kernel context plus (in most cases) the backing process context on systems implementing the combined user/kernel address space model.
- *The memory area holding the shellcode must be marked as executable*. In other words, the pages that hold the shellcode need to have the executable bit turned on. If you can place the shellcode in user land (which basically means you are targeting a local vulnerability in a combined address space environment), this is less of a problem, since you can easily set the mapping protections yourself. If your shellcode resides in kernel land, this may become more complicated.

- *In some situations, the memory area holding the shellcode must be in memory.* In other words, the kernel might implicitly consider the memory it is about to execute as paged in, so you cannot afford to make it take the shellcode page from disk. Luckily, your page will generally be paged in (in the end, you sort of recently accessed it to place the shellcode), regardless of whether you took care to explicitly handle it.

Let's now examine the different approaches to shellcode placement and how to overcome these constraints.

Shellcode in User Land

Anytime you can, *try to place your shellcode in user land*. Doing so affords a number of benefits.

First, it makes it easy to meet the requirements we listed in the preceding section, thereby allowing you to write robust exploits (exploits that will automatically detect if something has gone wrong and avoid crashing the machine), including exploits targeting local or remote vulnerabilities.

In a local vulnerability, you are the one triggering the vulnerability, and thus you have control over the user-land process that calls into the kernel. Mapping a portion of the address space with the privilege rights that you want is just as easy as correctly using the memory mapping primitives offered by the operating system. Even on systems that prevent a mapping to simultaneously be writable and executable (and prevent a previously writable segment from becoming executable during the lifetime of the process) you still can:

- Include the shellcode in the executable itself at compile/linking time. This implies that you can write the shellcode in C, a pretty nice advantage.
- Place your shellcode in a file and map that file, specifying executable permissions (and no writable ones).

You also get another advantage: you are not hampered by space constraints for the shellcode. In other words, you can make the shellcode as big as you want, and therefore you can add a large NOP landing zone on top of it. NOP landing zones greatly increase your chances of a successful exploitation, especially when you do not have full control over the address to which you will redirect the hijacked control flow.

For example, let's say you can control only the first part of the virtual address the kernel path will jump to, that is, the first 16 bits of a 32-bit address. That leaves 16 bits that can have any value. By mapping a memory area of 2^{16} bytes, filling it with NOPs, and placing your shellcode right after that, you ensure that no matter what value these 16 bits may assume, you will always execute what you want correctly, as Figure 3.3 shows.

As we stated previously, the ability to write shellcode in C is an interesting advantage. In fact, especially if you have a lot of recovery to perform, it is easier to write the logic correctly in C and let the compiler do the hard work for you, rather than to

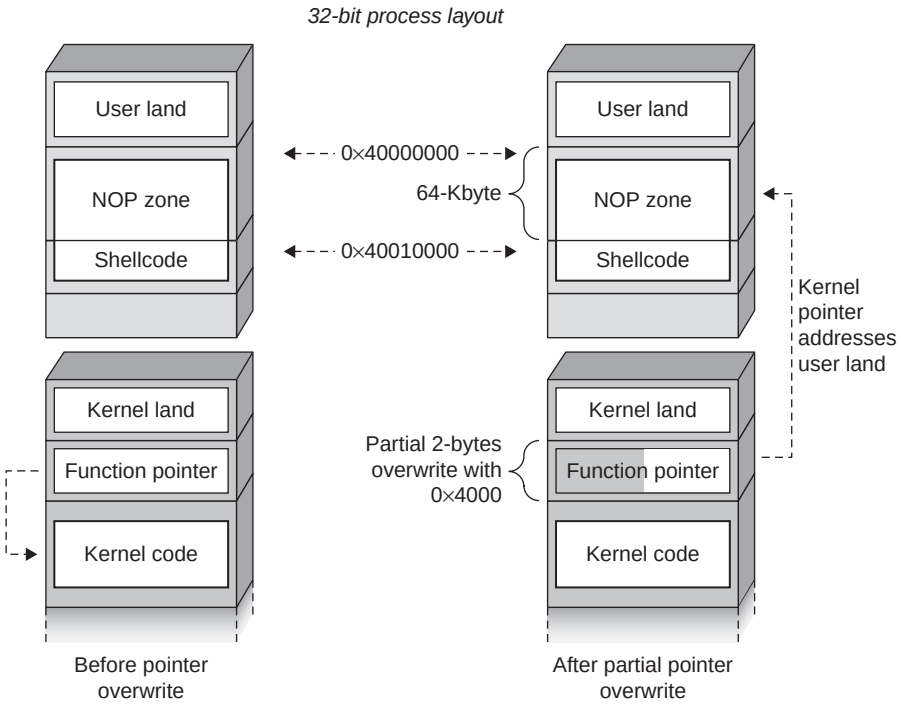

FIGURE 3.3

NOP landing zone on top of our shellcode.

churn out long assembly sequences. However, note that *the user-land code must be compiled with the same conventions the kernel is using*. In particular, the calling convention (which, as we said previously, might be affected by the compiler options) has to be respected, or you will just end up returning incorrectly from the function and panicking the machine. Also, you need to keep your code as self-contained as possible and avoid using functions in external libraries linked at runtime (or eventually, but not advised, compile the code statically). As an example, the x86-64 segment selectors are used differently in user land and kernel land, which means you would end up using a segment selector that is meaningful in user land from inside a kernel path with, again, the obvious panic outcome waiting around the corner.

Overriding the third of the previously stated constraints usually does not require any extra effort. If the shellcode is part of the exploit executable, it likely will be in the same pages used to run the executable and likely will not be evicted from memory before it is reached. In any case, you can also read a byte from inside the virtual addresses holding the shellcode to drive the kernel into bringing the specific pages in memory.

When ensuring that the shellcode is in the same context as the kernel path you depend on both the kernel memory model and the vulnerability. You cannot use

the user-land approach on a system where a user-land and kernel-land split is in place. In such a scenario, a user-land virtual address has a completely different meaning in kernel land.

To successfully reach the shellcode, you also need to be in the same execution context of the hijacked kernel path, to be sure that your process page tables are indeed the ones actively used in kernel land. Implicitly, that also means the user-land instructions right before the trap and those in the vulnerable kernel path have to execute on the same CPU. While in the context of a system call or of a synchronous interrupt "generated" by your code, this is always the case. However, if the vulnerable kernel path is inside an asynchronous interrupt handler or in a deferred procedure (i.e., helper routines that are scheduled to be executed at a later time and maybe on another CPU, in an SMP environment), all bets are off. In such cases (and in the case of a user/kernel address space split), you need to consider either a pure kernel space shellcode or, at least, a mixed/multistage approach.

Shellcodes in Kernel Land

If you cannot store the shellcode in user land, you need to store it in kernel land. However, life in kernel land is not as easy as it is in user land, and you need to overcome a couple of obstacles/issues:

- You have no control over the kernel page protections. You need to find a place that has already been mapped as executable and writable. This might not always be possible.
- You have a very limited view of the virtual addresses in kernel land. In other words, in the absence of an *infoleak*, you rely on the information that the kernel exports and that you can gather from user land, as we will discuss in the section "The Information-Gathering Step" later in this chapter.
- You usually do not have a way to directly write into kernel-land buffers, so you might need to find clever/original ways to make your shellcode appear in kernel land.
- Assuming that you found a memory area and that the area is under your control, you might be limited in the amount of space you can use. In other words, you need to be pretty careful about the size of the shellcode. Also, the shellcode most certainly needs to be written (and optimized) in assembly.

On the other hand, kernel page tables are obviously always visible from any executing kernel path (they are in the same context), and generally they are paged in (e.g., kernel code is locked in memory and operating systems explicitly indicate areas of the kernel as not pageable). We will discuss kernel-only shellcodes in more detail in Chapters 4 and 5.

Mixed/Multistage Shellcodes

Due to the usually limited size of kernel buffers and the advantages that user land offers, kernel-space-only shellcodes are not extremely common. A far more typical approach is to have a small stub in kernel land that sets up some sort of

communication channel with user land, or simply prepares to jump into a user-space shellcode. We call this kind of approach *mixed* or *multiple-stage shellcode*, to capture the fact that the execution flow jumps through various stages from kernel land to user land.

Mixed/multistage shellcodes are common when exploiting vulnerabilities triggered in an interrupt context, especially remote kernel vulnerabilities, where they are likely to trigger the bug inside the handler of the interrupts raised by the network card (we will discuss this in more detail in Chapters 7 and 8). The key idea here is that interrupt context is many things, but definitely not a friendly environment for execution. It should come with no surprise that kernel-level interrupt handlers are, usually, as small as possible.

NOTE

Although jumping to user land is the classic ending for such shellcodes, it is also possible to have a multistage shellcode that resides entirely at the kernel level. In such cases, we still prefer talking of multistage shellcodes (albeit not mixed) than of kernel-level-only shellcodes.

Let's now take a more detailed look at an example of a multistage shellcode. For simplicity, we'll consider a two-stage shellcode (but remember that more stages may have to/can be used):

1. The first thing the first stage needs to do is to find a place to store the second-level shellcode in the kernel. It can do this by allocating a new buffer or replacing static data at a known address. It is interesting to note that you were already able to start executing, and therefore you have a huge weapon in your arsenal: you can use the kernel subsystems and internal structures to find the memory areas you are interested in. For example, an advanced shellcode can go through the list of active processes and look for one listening on a socket, or read through the kernel list of symbols and resolve the address of important system structures such as the system call table.
2. After the second stage has been placed somewhere in the kernel, the first stage needs to transfer control to it. With this operation you can escape from interrupt context, if you need to. As an example, after finding the system call table in the preceding step, you can replace the address of a frequently used system call and just wait for a process to trigger it. At that point, your code will execute in the much more comfortable *process context*.

Mixed shellcodes meet the constraints we introduced at the beginning of this section in the same way as their user or kernel space counterparts do, depending on where the stage that is about to execute resides. As you will see in Part III of this book, when we discuss remote kernel exploitation, a three-stage approach is generally the way to go. The first stage sets up the transition to process context,

and the second stage modifies some user-land program address space and then jumps into executing the third-stage shellcode in user land (socket primitives are a lot easier to code in user land).

Return to Kernel Text

We will end our analysis with a particular kind of kernel space shellcode that you can use to bypass advanced kernel protections that prevent you from finding a suitable writable and executable area for your shellcode. The technique we're presenting here overcomes this issue by creating a shellcode that does not contain any instruction, but instead contains addresses and values. Such a shellcode does not need to be stored inside any executable area. If you are familiar with user-land exploitation, this approach is a close relative of both the *return into lib* and *code borrowing* techniques for bypassing nonexecutable memory protections.

The first catch regarding these techniques is that at least one place must be mapped as executable: the memory mappings that compose the executable itself! In user land, that means the binary and all the dynamic libraries it uses. In kernel land, it refers to the kernel and all the code segments of the loaded modules (if a modular kernel is used). The second catch is that you could find chunks of instructions inside the executable mappings that, if chained together/used correctly, may lead to an increase in privileges.

This kind of approach is tightly linked to (and dependent on) the underlying architecture, the ABI, and even the compiler. In particular, we are interested in the calling convention in use (i.e., where is the return address saved, and how are parameters passed?).

TIP

On the x86/x86-64 architecture, instructions are variable in size, and you are allowed to start executing from any address—even in the middle of a particular instruction—and have the stream of bytes interpreted starting from there. This is usually exploited to find short sequences. For example:

```
a) bb 5b c3 ff ff      mov    $0xffffc35b,%ebx
b) 5b                  pop    %ebx
   c3                  ret
```

By jumping one byte after the start of the `mov` opcode, we actually get to a *pop %ebx; ret* sequence, even if those two instructions are not used one after the other in the kernel. Note that we do not bother to have valid instructions after the `ret`; the control flow will be transferred before reaching valid instructions after the `ret`. On RISC architectures, instructions are fixed in size, and jumping to addresses not aligned to the instruction size results in an error. Basically, you cannot jump in the middle of an instruction to have it interpreted differently.

Return addresses among the various procedures are commonly saved on the stack; thus, in most situations, stack control is mandatory for the success of this technique. The classic scenario is a stack overflow that allows you to overwrite the

return address and, if the ABI dictates that parameters are passed on the stack (as is the case on x86 32-bit systems), lets you forge a controlled set of parameters for the target function. At that point, you have a variety of options, depending on the following:

- What the vulnerability allows you to do. In other words, how much stack space can you overwrite and how much control do you have on the values you write?
- What the architecture allows you to do. Here is where the ABI and, eventually, the compiler get into the game. If the parameters to the function get passed on the stack, you need more stack space, but you have a greater deal of control over what the function will use. If they are passed on registers, you need to get the registers filled with proper values somehow, but you may end up using less space on the stack.

Assuming full and arbitrary control on the stack and stack-based parameter passing, you create a shellcode made of a mix of function addresses, parameters, and placeholder space (to accommodate the architectural use of the stack) that would do the following:

- Use a kernel function that allocates some space marked as executable.
- Chain a kernel function to copy a set of bytes from user land (or from some known kernel area) into the previously returned address.
- Leave the last return address so that the code will jump into the chosen memory address.

The copied-in code starts executing, and from that moment on you are in a traditional kernel shellcode scenario.

As you can imagine, this approach gets increasingly complicated as you stack in more functions. For those of you who are familiar with user-land exploitation, this approach can be seen as a kernel-level return into lib.

Fortunately, a different approach is available, since you are not obligated to return to the entry point of a function. Since we assumed full knowledge of the kernel code address space (which is not an unlikely scenario, as you will see in more detail in the section "The Information-Gathering Step"), you can look for a chunk of instructions that will do something useful. As an example of this, think about the privilege system in use on your OS: Most likely, there is a kernel function (even a kernel system call) that allows a privileged process to reduce or elevate its privileges. This function will probably receive the new process privilege value as a parameter, do a bunch of checks on the process making the call (obviously, an unprivileged process cannot raise its own privileges), and then get to some code that will just copy the new value over the process's stored credentials. Regardless of the architecture and compiler options, the new credentials will end up in a register, since it is accessed multiple times (to check it against the current process, to check if it is a privileged request, and, at the end, to eventually set the value in the process credential structure).

At this point, you can do one of the following:

- Drive the setting inside the register of the highest privilege level value. Since you control the stack, this is less complicated than it may sound. All you have to do is find some code that pops the content of the stack into the register and then issues a return call (which, again, generally just pops a value from the stack and uses it as the return value). Even if the specific sequence is never used in the kernel, on a non-RISC architecture you may still find it somewhere in memory, as we mentioned in the previous Tip box.

TIP

Zero is a typical value for indicating high privileges (when represented by an integer) and *0xFFFFFFFF* is a typical value when the privilege set is represented by a bit mask. Both of these values are pretty common inside a function (e.g., –1 is a classic way to indicate an error and 0 is a classic way to represent success). The odds of not having to set the register (and therefore bypass the first step we just described) are not always that bad…

- Place the return address on the stack and make it point inside the privilege setting function, right after the checks.
- Prepare a fake stack frame to correctly return to user land. In fact, since you are not using any specific kernel-level shellcode (as you were doing in the previous example), you need to provide a clean way to get out from the kernel. This depends on the way you *entered* the kernel in the first place and, again, is highly ABI-dependent.

This second approach we just described is similar to the code borrowing technique. If you are interested in these user-land techniques (e.g., if you are looking for a detailed explanation or more ideas for bringing them into kernel land), interesting resources are listed in the "Related Reading" section at the end of this chapter.

Forging the Shellcode

Now that we have extensively covered placing the shellcode, it is time to discuss what operations it should perform. As we said at the beginning of this section, a good shellcode needs to do at least two things: gain elevated privileges and recover the kernel state. There are many different ways to perform the privilege escalation task, and some of them can be pretty exotic, including creating gateways inside the main kernel structures to open backdoors that can be used later to modify the kernel page tables to allow direct access from user land, or changing the path of some user-land helper program. We will focus here on the most common method: modifying the process credentials stored in the process control block.

TIP

When you are targeting a hardened environment, since the shellcode executes with full privileges, it is usually a good idea to disable eventual security restrictions (e.g., escape from a confined environment such as a FreeBSD jail or a Solaris zone) or disable security protections (e.g., shut down SELinux on a Linux kernel).

Raising Credentials

Raising credentials is the most common task that almost all local privilege escalation exploits perform. Credentials are kept in one or more structures contained in the process control block and they describe what a process is allowed to do. Storing credentials can be as simple as an integer value identifying the user, as in the traditional UNIX root/generic user model, or representing a whole set of privileges or security tokens, as is usually the case when a role-based access control system and the least privilege model are in place (tokens are the typical privilege model on Windows). Different operating systems use different authentication and authorization models, but most of the time the sequence that leads to a certain user being authorized or denied a set of operations can be summarized in the following steps:

1. The user authenticates itself on the system (e.g., through the classic login/password mechanism).
2. The system gives the user a set of security credentials.
3. The authorization subsystem uses these credentials to validate any further operation that the user performs.

After the user has correctly logged in (the authentication phase), the kernel dynamically builds the series of structures that holds information related to the security credentials assigned to the user. Every new process spawned by the user will inherit the aforementioned credentials, unless the user specifies differently (the operating system always provides a way to restrict the set of privileges at process creation time). Whenever a process wants to perform an operation, the kernel matches the specific request with the stored set of credentials and either executes the operation on top of the process or returns an error.

The goal of the shellcode is to modify those credentials so that an extended set of privileges is granted to your user/process. Since the credential structures are stored inside the process control block, it is usually quite easy to reach them from inside your shellcode. There are two main ways to identify the correct values to change:

- You can use *fixed/hardcoded offsets* and perform very simple safety checks before using them. For example, if you need to dereference a pointer to reach a structure, you would just check that the address you are about to dereference is within the kernel-land address space.

- You can use a *heuristic approach*. Credential structures have a precise layout in memory, and you know what credentials you were granted. Based on that, you perform a pattern match in memory to find the correct values to change. Relative offsets inside a structure may change, and using this heuristic approach you can figure out the correct place at runtime.

In general, a hybrid approach can be used against nearly all kernels, identifying the offsets that have been constant over the years and using more or less sophisticated heuristics to derive the other ones. A typical and effective heuristic is to look for specific signatures of structure members that you can predict. For example, a process-based reference counter would have an upper bound value with the number of processes (easy to check), or in a combined environment a kernel address will always have a value higher (or lower, depending on where the kernel is placed) than the split address.

Recovering the Kernel State

Gaining full privileges on a machine is exciting; losing them after a second due to a kernel panic is a lot less fun. The *recovery phase* aims to extend the fun and keep the machine up and running while you enjoy your freshly gained privileges. During the recovery phase you need to take into account the following two issues:

- The exploit may have disrupted sensible kernel structures and, in general, trashed kernel memory that other kernel paths may need to access.
- The hijacked kernel control path may have acquired locks that need to be released.

The first issue primarily concerns memory corruption bugs. Unfortunately, when you exploit memory bugs, you cannot be very selective. Everything between the buffer that you overflow and your target will be overwritten, and in many cases, you do not have enough control of the overflowing size to stop exactly after your target. In this case, you have two different types of structures to recover: stack frames and heap control structures.

> **NOTE**
>
> In most architectures/ABIs, stack frames are deeply involved in procedure chaining and software traps. Although we have tried to keep the following discussion as generic as possible, in order to appreciate the details of stack recovery we actually need to focus on a specific architecture implementation. Since our architecture of choice is x86-64, each practical part that follows in this subsection is based on the x86-64 implementation.

During a stack-based memory overflow you may or may not be able to get back to a sane state. For instance, you might be able to tweak the shellcode to return to one of the nested callers of the vulnerable path and continue the execution from there.

However, if you have trashed far too much stack, you'll need to terminate the function chain and jump back to user land. As you already know, user-land processes reach kernel land through a software trap/interrupt. Once the kernel has finished performing the requested service, it has to return control to the process and restore its state so that it can continue from the next instruction after the software trap. The common way to get back from an interrupt is to use the *IRETQ* instruction (*IRET* on x86). This instruction is used to return from a variety of situations, but we are interested here in what the Intel Manuals call *inter-privilege return*, since we are going from kernel land (the highest privilege level) to user land (the lowest privilege level).

The first operation that the *IRETQ* instruction performs, shown here in the pseudocode syntax used in the Intel Manuals, is to pop a set of values from the stack:

```
tempRIP ← Pop();
tempCS ← Pop();
tempEFLAGS ← Pop();
tempRSP ← Pop();
tempSS ← Pop();
```

As you can see, *RIP* (the 64-bit instruction pointer), *CS* (the code segment selector), *EFLAGS* (the register holding various state information), *RSP* (the 64-bit stack pointer), and *SS* (the stack segment selector) are copied in temporary values from the stack. The privilege level contained in the *CS* segment selector is checked against the current privilege level to decide what checks need to be performed on the various temporary values and how *EFLAGS* should be restored. Understanding the checks is important to understanding what values the architecture expects to find on the stack. In our case, the *CS* holds a lower privilege level (returning to user land), so the registers on the stack need to contain the following:

- ***CS, SS*** Respectively, the code and the stack segment used in user land. Each kernel defines these statically.
- ***RIP*** A pointer to a valid executable area in kernel land. Our best choice here is to set it to a function inside our user-land exploit.
- ***EFLAGS*** Can be any valid user-land value. We can simply use the value that the register has when we start executing our exploit.
- ***RSP*** A pointer to a valid stack, which can be any amount of memory big enough to allow the routine pointed to by *RIP* to safely execute up to the execution of a local shell with high privileges.

If we prepare the values of these registers correctly, copy them in memory in the order that *IRETQ* expects, and make the kernel stack pointer point to the aforementioned memory area, we can simply execute the *IRETQ* instruction and we will get safely out of kernel land. Since the stack contents are discarded at each entry to kernel land (basically, the stack pointer is reset to a fixed value offset from the start of the page allocated for the stack, and all the contents are considered dead), that is enough to safely keep the system in a stable state. If the

kernel and user land take advantage of the *GS* selector (as is done nowadays), the *SWAPGS* instruction needs to be executed before *IRETQ*. This instruction simply swaps the contents of the *GS* register with a value contained in one of the machine-specific registers (MSRs). The kernel did that on entry, and we need to do that on the way out. As a quick recap, the stack recovery phase of our shellcode should look like this:

```
push    $SS_USER_VALUE
push    $USERLAND_STACK
push    $USERLAND_EFLAGS
push    $CS_USER_VALUE
push    $USERLAND_FUNCTION_ADDRESS
swapgs
iretq
```

Because heap structure recovery depends on the operating system implementation and not on the underlying architecture, we will discuss it in detail in Chapters 4, 5, and 6. For now, though, it's important to know that unless some sort of heap debugging is in place, overwriting allocated heap objects does not require a lot of recovery (usually just enough emulation of valid kernel values to let the kernel path using them reach the point where they free the object). Overwriting free objects instead might require some more handling, since some kernel heap allocators store management data inside them (e.g., the "next" free object). At that point, having been able to drive the heap into a predictable state is of great help, and we will discuss the theory behind achieving such a result in the following section, "The Triggering Step."

So far we have focused on recovering from problems created after the vulnerability has been triggered. We have paid almost no attention to what the kernel path has done before reaching the vulnerability and what it would have done if the execution flow hadn't been hijacked. In particular, we need to be especially careful to release eventual resource locks that might have been acquired. For vulnerabilities that add execution blocks, this is not an issue. Once done with our shellcode, we will return exactly after the hijacking point and the kernel path will simply finish its execution, clearing and releasing any resource it might have locked.

On the other hand, disruptive hijacks such as stack overflows using the `IRETQ` technique described earlier never return to the original kernel path, so we need to take care of locks inside the shellcode during the recovery phase. Operating systems implement a variety of locking mechanisms: spinlocks, semaphores, conditional variables, and mutexes in various flavors of multiple/single readers/writers, to name a few. This variety should not come as a surprise: locks are a critical performance point, especially when a resource is contended by many processes/subsystems. We can divide locking primitives into two main parts: *busy-waiting locks* and *blocking locks*. With busy-waiting locks the kernel path keeps spinning around the lock, cranking CPU cycles and executing a tight loop until the lock is released. With blocking locks, if the lock is already held,

the kernel path goes to sleep, forcing a reschedule of the CPU and never competing for it until the kernel notices that the resource is available again and wakes the task back up.

The first thing you need to do when you write an exploit that will disrupt execution flow is to identify how many critical locks the kernel path acquires and properly release each of them. A critical lock is either one on which the system depends (there are just a handful of those in each operating system, and they are generally spinlocks), or one that drives to a deadlock in a resource that you need after the exploit. Some kernel paths also perform sanity checks on some locks; you must be careful to not trap/panic on one of those, too. All critical locks need to be restored immediately.

On the other hand, noncritical locks can be either fixed indirectly at a later stage (e.g., loading an external module) or just forgotten if the unique effect is to kill the user-land process (it is as easy to raise the parent process credentials as it is to raise the current process ones), or to leave some noncritical resource unusable forever.

THE TRIGGERING STEP

Now that we have a working shellcode placed somewhere in the kernel it is time to start creating the conditions to reliably reach it. This is the job of the triggering step.

Our main goal here is to create the conditions for a successful hijacking of the kernel execution flow. Leaving aside those logical bugs that do not involve arbitrary code execution, we'll divide the analysis of this phase into two main categories: *memory corruption* issues and *race conditions*.

Memory Corruption

As you saw in Chapter 2, there are different types of memory corruption, but our final goal is always to overwrite some pointer in memory that will be used later as an *instruction pointer* (i.e., it will end up in the PC/IP of the CPU). This can be done either directly, by overwriting the return address of a function placed in the kernel mode stack, or indirectly, by emulating one or more kernel space structures until we are able to reach a kernel path using our controlled function pointer. Following the distinction we made during our taxonomy, we'll now evaluate the three common cases of memory corruption: *arbitrary memory overwrite*, *heap memory corruption*, and *stack memory corruption*.

Arbitrary Memory Overwrite

Arbitrary memory overwrite is a fairly common scenario in kernel land. In this situation, you can overwrite arbitrary memory with either (partially) controlled or uncontrolled data. On nearly all current operating systems/architectures, read-only

sections are protected from privileged direct writing. On the x86 and x86-64 architectures, this is the job of the *WP* flag, which we can take for granted as being set. Our goal is thus to find some writable place that, once modified, will lead to the execution of our code.

Overwriting Global Structures' Function Pointers

Earlier in this chapter, we mentioned the possibility of overwriting function pointers stored in kernel structures. The usual problem with this approach is that most of these structures are dynamically allocated and we do not know where to find them in memory. Luckily, nearly all the kernels need to keep some global structures.

WARNING

If global structures get declared as constant (with *const* being the typical C keyword for that), the compiler/linker will place them in the read-only data section, and if this section's mapping flags are honored, they are no longer modifiable. On the other hand, if they need to change at runtime, they have to be placed in a writable segment. This is exactly the kind of entry point we are looking for.

A typical C declaration of a struct holding function pointers looks like this:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
                      size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                         unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                          unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *,
                  unsigned int, unsigned long);
    [...]
```

The preceding example is taken from the Linux kernel and is used to create an abstraction layer between the filesystem-specific code and the rest of the kernel. Such an approach is pretty common in modern operating systems and it generally provides a very good entry point for hijacking the execution flow. As you will see in the section "The Information-Gathering Step," it may be extremely easy (and reliable) to locate these structures in memory. If you are looking for this kind of structure for your exploit, just hunt for type identifiers containing the *ops* or *operations* name in your operating system of choice.

Exploiting the Architecture

We started this chapter with an analysis of the architecture level. Apart from being the base from which to properly understand the low-level details of the execution phase (and the low-level details of the operating system), the architecture can turn into an ally and offer new exploitation vectors. Earlier, we mentioned interruptions and exceptions and the fact that the operating system registers a table of pointers to their handlers. Obviously, if you can modify such pointers, you can hijack the control flow and divert it toward your shellcode.

As an example, let's consider the IDT from the x86-64 architecture. Figure 3.4 depicts an entry in this table.

As you can see in Figure 3.4, the entry is 16 bytes long and is composed of a number of fields:

- **A 16-bit code segment selector** This indicates the segment selector for the kernel interrupt handler. Usually, it holds the kernel code segment selector in which the routine resides. Basically, this field specifies the selector to use once the handler function gets called.
- **A 64-bit offset for the instruction pointer (RIP)** This specifies the address to which the execution will be transferred. Since 64 bits are used, that allows an interrupt service routine to be located anywhere in the linear address space.
- **A 3-bit interrupt stack table (IST)** The stack switching mechanism uses this between privilege levels. This field was introduced in the x86-64 architecture to

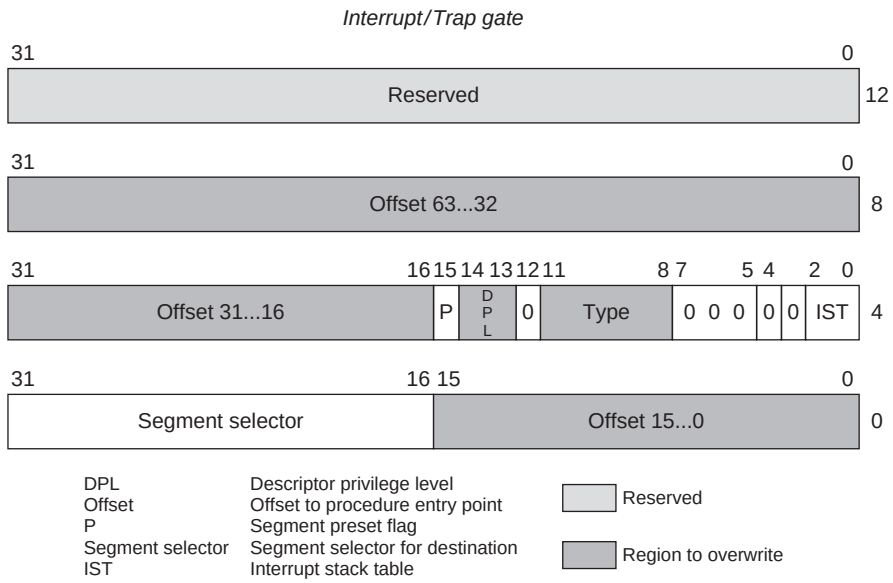

FIGURE 3.4

An x86-64 interrupt/trap gate entry.

provide a means for particular interrupts to use a known good stack when executed. This is usually not the case for the kind of interrupt we are aiming to modify, so we can ignore/disable it. You can find more about the IST and the stack switching mechanisms in the manuals referenced in the "Related Reading" section at the end of this chapter.

- **A 4-bit type that describes the descriptor type** There are mainly three types of IDT descriptors: task gates, interrupt gates, and trap sates. We care only about interrupt and trap gates, since corruption of a task gate does not directly lead to arbitrary execution. Interrupt gates are used to serve external hardware interrupt requests, while trap gates are usually used to service exceptions and software-generated interrupts (e.g., the one created by the *INT* instruction).
- **A 2-bit *DPL* (descriptor privilege level) field** This field is compared against the caller *CPL* (current privilege level) to decide if the caller is permitted to call this gate.
- **A 1-bit *P* (present) flag** This indicates if the segment is present or not.

To insert a new kernel gate under our control, we can simply replace an entry of choice. Actually, in case the vulnerability does not allow us to or to simplify the operation, we can achieve the same result by selectively overwriting only part of the IDT entry, the *DPL* and the *RIP OFFSET* values. We need to set the *DPL* value to the binary value *11* (three), to specify that unprivileged user-land code (running with *CPL = 3*) is allowed to call the gate handler. Also, we need to modify the *RIP OFFSET* value to point to our user-land routine. The easiest way to do this on a combined user/address space model is to simply pick a user space routine and write its address in the various *OFFSET* fields. Since we control the user-land address space, though, we can also modify a few of the most significant bytes of the address and make it point somewhere below the kernel/user space split address. Note that in such a case we do not have full control over the address value, and to successfully transfer control to our routine we may have to use, for example, a NOP-based technique such as the one we described earlier in the "Placing the Shellcode" subsection.

Heap Memory Corruption

The majority of kernel temporary buffers and data structures get allocated in the kernel heap. As usual, performance is a key factor in their design, as the allocation and relinquishment of heap objects has to be as efficient as possible. For this reason, as you saw in Chapter 2, extra security checks (e.g., to detect an overflow of the heap object) are usually turned off on production systems. We also already discussed the ideas on which the heap allocator is based. What we are interested in now is if and how we can influence its behavior and what we can do when we generate an overflow.

Controlling the Heap Allocator's Behavior

A user mode process cannot directly interact with the kernel heap allocator, but it can nonetheless drive the allocation of different types of heap-based objects,

just invoking different system calls. A typical kernel offers hundreds of system calls with a variety of options. Let's return to the earlier filesystem example: A user process opening a file forces the allocation of a kernel structure to keep track of the file being opened. This structure (and, potentially, other structures connected to this one) needs to be allocated from the heap. By opening thousands of files and then releasing them, a user-land process can grow and shrink the kernel heap in a more or less controlled fashion. But why is that important?

The heap allocator usually allocates and frees objects in a (somehow) predictable way. Usually the process works in one of the following ways:

- A free list for each generic size/type of object is maintained. Each time an object is freed it is attached to the list (either on top or at the bottom). Each time an object is requested the first object on the list is returned. The typical free-list implementation uses a LIFO approach, which means the last freed object will be the one returned in the next allocation.
- Each free object maintains a pointer to the next free object within itself, and the metadata handling the cache holds a pointer to the next free object. To avoid confusion, we call the first pointer the *object-pointer* and the second pointer the *cache-pointer*. At each point in time, there are as many object-pointers as there are free objects (each object holding the address of the next free object and the last one holding some termination value), and a single cache-pointer, holding the address of the next free object that will be returned. Whenever an object is requested, the cache-pointer is evaluated; the object it specifies is marked as being in use and is then returned. The selected object-pointer value is stored in the cache-pointer. Each time an object is freed, its object-pointer is updated with the address stored in the cache-pointer and its address becomes the new value of the cache-pointer.

At some point during its lifetime, the allocator will run out of free objects. In that case, a new page is allocated from the physical allocator and is divided into objects that will then either populate the free list (if the first type of allocator is in place) or initialize each one with the address of the *next one* and mark it as free (if the second type of allocator is in place).

As you can imagine, though, objects are not freed in the same order they are allocated, which means the free objects are not contiguous in memory. Since the list of free objects affects the address of the objects that get allocated, after some time subsequently allocated objects will not be contiguous in memory. The typical heap layout of a running system is thus fragmented, as shown in Figure 3.5. Although Figure 3.5 depicts the state of one cache, the same principle applies to all the various caches in the system.

As we noted earlier, you can drive the allocation of a large number of equally sized objects. This means you can fill the cache and force it to allocate a new page. When a new page is allocated, the position of the next allocated object relative to a specific object is generally quite predictable. This is

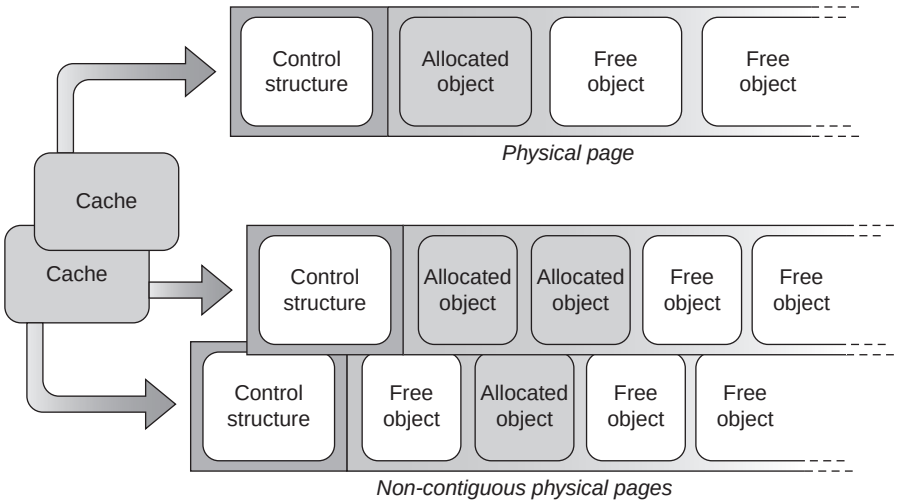

FIGURE 3.5

A fragmented heap layout.

exactly what we aim for to carry out our attack. Unfortunately, life is not quite that easy:

- To optimize performance, allocators may have many more variables that affect them. As a classic example, on an SMP system, for performance reasons the address of an object may also depend on the processor that runs when the allocation is requested, and we may not have control of that. This property is usually defined as its *locality*.
- Doing a specific system call also affects other parts of the system, which in turn might affect the behavior of the heap allocator. For example, opening thousands of files might require spawning more than a single thread, which in turn would force the allocation of other, different objects. We have to study this carefully to precisely understand the various interactions.
- We need to find a kernel path that opens an object and keeps it open until we decide to close it. Many paths allocate objects for the lifetime of the syscall and free them upon returning. Those paths are mainly useless for our purposes. On the other hand, some paths might depend on a user-passed option for the size to allocate. Those paths are pretty useful for filling different caches easily.

Heap Overflow Exploiting Techniques

We know we can somehow control the heap layout and force the allocation of an object in a specific place. Although we do not know the virtual address of this place, we can be more or less sure (depending on the degree of control we have over the allocator) about its position relative to other objects in memory, cache

metadata information, and other pages in the physical address range. Exploiting the heap involves using the best out of these three scenarios, which we will now describe in more detail.

Overwriting the Adjacent Object

This is the most used and reliable technique, and it works (with adjustments) on nearly any heap allocator. It basically involves overwriting the object adjacent to the overflowing object. If you recall the example we provided in the "Controlling Heap Allocator's Behavior" subsection, it basically means to overflow into C by writing past A. For this technique to be successful, C needs to have some sensitive information inside it. The obvious (and ideal) option is for C to hold either a function pointer so that we end in the case we described in the "Overwriting Global Structures' Function Pointers" subsection, or a data pointer that later will be used in a write operation so that we end in the case we described in the "Arbitrary Memory Overwrite" section.

TIP

Although looking for a function pointer is the classic approach, it is by no means the only option. You could look for a variable used as a size in a following allocation, a reference counter, or a lock to manipulate, among many other options. You are limited only by your imagination.

The steps to trigger such a scenario (in the common LIFO free objects situation) are as follows:

1. Force the allocation of a new page for the cache.
2. Allocate a placeholder object.
3. Allocate the target object.
4. Free the placeholder object.
5. Allocate the victim object.
6. Trigger the vulnerability (e.g., a buffer overflow) over the victim object, to overwrite the target object.
7. Force the execution out of the target object.
8. (Eventually) perform the necessary recovery as a consequence of the previous overwriting.

If the cache is not implemented with a LIFO approach for free lists, you need to substitute steps 2–5 with whatever algorithm is necessary to have two adjacent objects so that your victim object gets allocated once the target object has *already* been allocated. If allocating an object and triggering the overflow over it are two *decoupled* operations (i.e., if you can hold a reference and decide at what point in time to generate the overflow), the placeholder object becomes unnecessary. Figure 3.6 shows an example of this kind of approach.

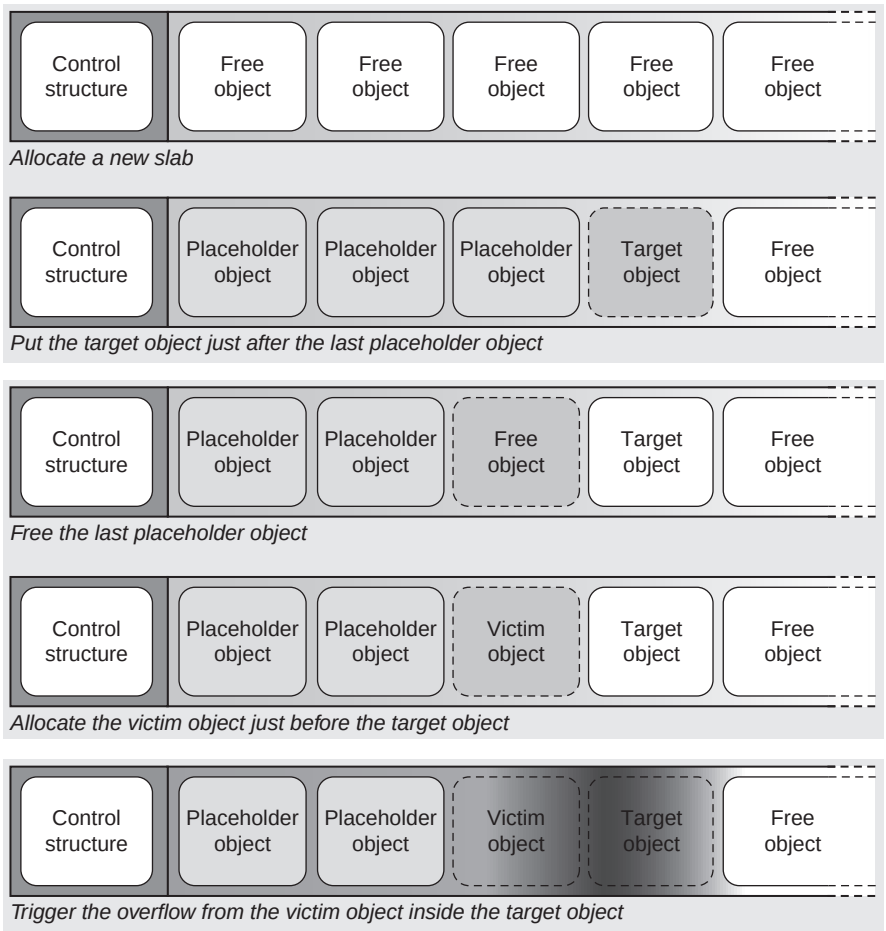

FIGURE 3.6

Overwriting the adjacent object technique.

Overwriting Controlling Structures

A few heap allocator implementations make use of *in-cache* and even *in-object controlling structures.* In such a case, we have a new attack vector that is based on overwriting sensible members of those controlling structures. Let's take a closer look at them, starting with the in-cache structure.

The in-cache structure may reside at the end or at the beginning of each page allocated to hold objects. If the structure is at the beginning of the page, there is really little you can do, unless you are lucky enough to hit a *buffer underflow* (write before the content of the buffer, for example, as a consequence of a negative offset) of the object. We will discuss another option for this situation in the

section "Overwriting the Adjacent Page." For now, let's focus on an in-cache controlling structure that is at the end of the allocated page.

Such a structure holds a variety of members describing the cache. The type and position of those members vary among operating systems, but a couple of them are nearly always present:

- The name of the cache or some similar identifier
- A pointer to the next free object
- The number of objects in the cache
- (Eventually) constructor and destructor functions to be invoked at object creation/release (to see how this can be useful, consider that a destructor function adds a lot of overhead, so you might want to use it on a cache basis)

This is by no means an exhaustive list of the potential members, but it does show a couple of interesting entry points:

- Overwriting the next free object pointer might allow you to drive the allocator into using/modifying memory under your control.
- Overwriting the constructor/destructor pointers (if present) might directly lead to code execution (in a fashion similar to what we explained in the "Overwriting Global Structures' Function Pointers" subsection).
- Changing the number of objects in the cache might result in some funny allocator behavior (e.g., trying to gather statistics from memory areas that are not part of the cache, and turning into a sort of *infoleak*).

We are considering more than one vector of exploitation, instead of picking one and just living happily with it, because in some situations we might end up with an overflow of only a few bytes and be unable to reach all the way down to our member of choice.

Now that you have a fairly clear idea of what to overwrite, here are the steps to do it:

1. Exhaust the cache so that a new page is allocated.
2. Calculate the number n of objects that compose the cache.
3. Allocate $n - 1$ objects.
4. Allocate the victim object.
5. Overflow into the in-cache controlling structure.

The approach can be visualized in Figure 3.7.

An example of in-cache controlling structure implementation is the FreeBSD Unified Memory Allocator, and a detailed article on its exploitation, "Exploiting UMA, FreeBSD kernel heap exploits," was released in PHRACK 66 by argp and karl.

The second type of controlling structure we will evaluate resides in the free objects and is generally used to speed up the lookup operation to find a free object. Such an implementation is used in the Linux SLUB allocator, and we will discuss it in detail in Chapter 4. The exploit that we will show there is also a good

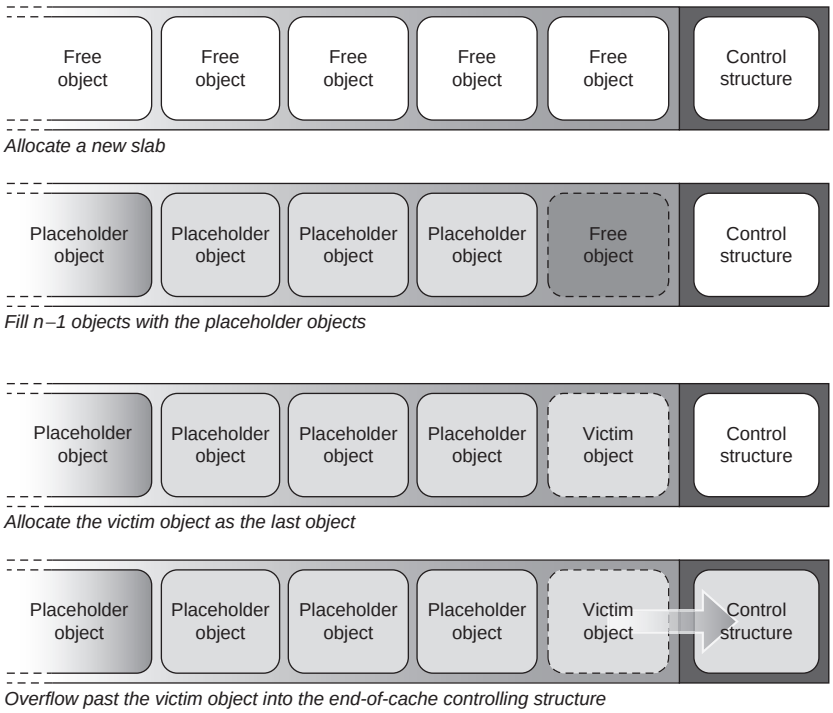

FIGURE 3.7

Overflowing into the cache controlling structure.

example of an overflow of a small number of bytes (actually, a single byte overflow, generally known as *off-by-one*… yes, there is a bit of magic in that exploit).

This type of controlling structure varies a lot, depending on the allocator implementation, and so it is hard to present a general technique. The idea we want to highlight here is that even a single byte, if correctly tweaked, can lead to a full compromise.

Overwriting the Adjacent Page

Let's say you have a heap overflow, but no object in the specific cache holds any sensible or interesting data. Moreover, the controlling structure is kept *off-slab* or is at the start of the cache, and thus is unreachable. You still have a shot at turning the heap overflow into a successful compromise: *the physical page allocator*.

The technique we are about to present is valid in *any* operating system, but is definitely less reliable than the two previous ones, because it involves an extra subsystem beyond the heap allocator. In particular, it involves the subsystem the heap allocator depends on: the physical page allocator. When we first described

a generic heap allocator, we said that it is a consumer of the physical page allocator from which it receives physical pages that it then divides into objects and manages internally. Virtually any other area of the kernel that needs memory ends up using the physical page allocator; from the filesystem page cache to the loading of modules, at the very bottom it is all a matter of populating pages of memory. And memory, as you know, is contiguous. If you take a picture of a computer's physical memory at a given time, you see a list of potentially independent pages sitting next to each other. Scattered among those pages are the heap allocator pages, and it is exactly that condition that gives you a new attack vector.

The idea is pretty simple: you place the *victim* object at the very end of the cache, and from there you overflow into the next *adjacent* page. The main problem is predicting with some degree of precision what will be after your page, and also managing to place a sensible structure there. Controlling the physical page allocator from user land is challenging. Although operating systems usually export some degree of information about the heap allocator, they provide a lot less information about the physical allocator. Moreover, each operation you perform to drive the allocation of a new page likely will have side effects on the page allocator, disturbing the precision of your algorithm; the same thing happens with any other unrelated process running on the system (a few extra unexpected page faults might invalidate your layout construction just enough to miss your target). Note that here you are trying to have two pages next to each other in memory.

One way to improve your chances is to rely on a sort of probabilistic approach:

1. Exhaust the *victim object cache* up to the point where all the available objects are allocated, but a new empty page is not. That might involve taking care of specific thresholds that the allocator might impose to proactively ask for pages to the physical allocator.
2. Drive the allocation of tons of pages, exhausting the number of free pages, by requesting a specific resource (e.g., opening a file). The aim is to get to a situation such as the one depicted in Figure 3.8a.The fewer side effects the allocation has (as a rule of thumb, the less deep a kernel path goes to satisfy the request), the better your chances of success. A link between this resource and the victim object is not necessary. It is only important that this specific resource puts some controlling structure/interesting pointer at the beginning of the page (the closer it is to the beginning, the smaller the number of variables trashed during the overflow that you need to emulate/restore).
3. Free some of the resources you allocated midway through the process so that the amount of freed memory adds up to a page. Since the kernel is under memory pressure (you generated it in the previous step), the page will be returned to the allocator immediately and will not be cached or "kept" by whatever subsystem you used during the exhaust phase. The catch here is to

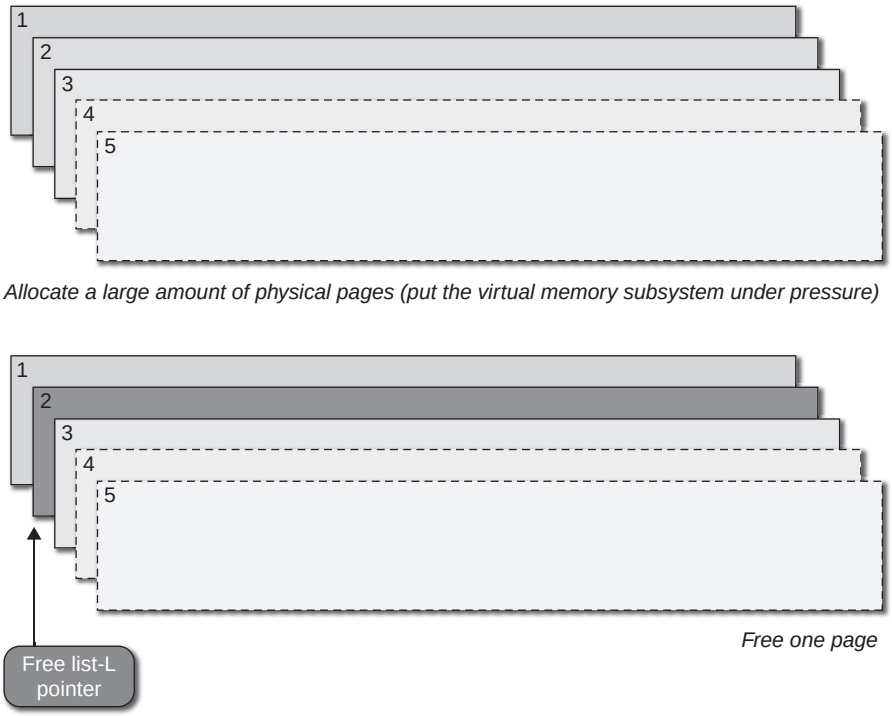

FIGURE 3.8a

Driving the allocation of multiple pages and freeing one of them.

free some of the early allocated resources so that the freed page lies physically between some of the pages holding the resource you are targeting (as shown in Figure 3.8a).

4. Drive the allocation of a new page for the victim object cache by allocating a few more objects. The freed page will be returned to the heap allocator.
5. Perform the overflow from the victim object over the next adjacent page.
6. Start freeing, one after the other, all the resources you allocated during the physical page allocator exhaust phase, hoping that one of them has been overwritten by the overflow of the previous step.

The last steps of this approach are shown graphically in Figure 3.8b.

As you can imagine, there is the risk of overwriting a wrong page, and thus touching some sensible kernel data. In that case, the machine will panic and your target will be lost. This is another reason why limiting the number of overflowed bytes as much as possible is important.

On a machine with a low load, this technique can be implemented rather efficiently. We will discuss this in more detail in Chapter 4.

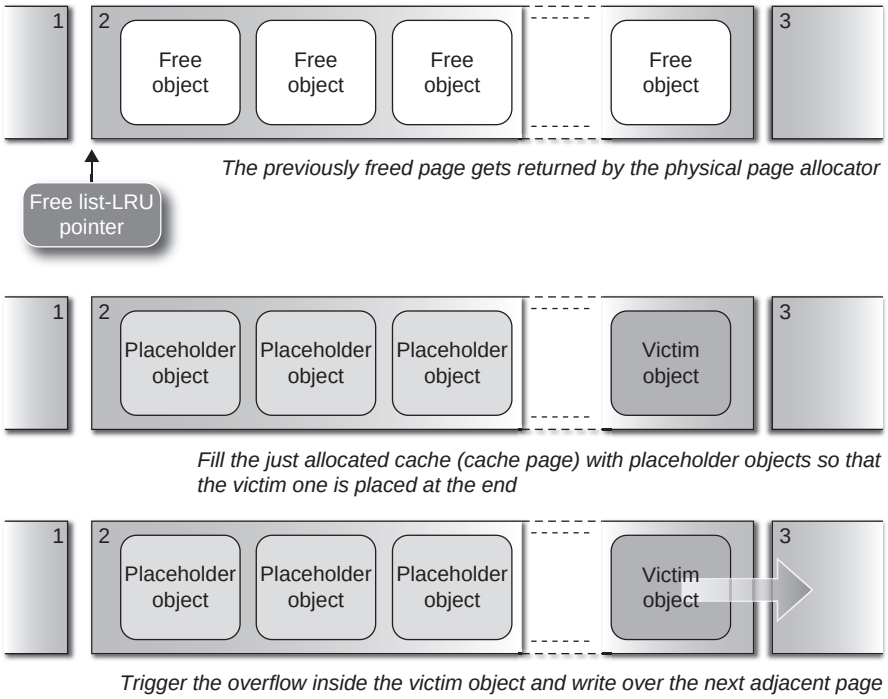

FIGURE 3.8b

Overflowing into the adjacent page.

Kernel Stack Corruption

As we mentioned in Chapter 2, each user-mode application has at least two stacks: a user-mode stack and a kernel-mode stack. In this section, we'll focus on techniques you can use when an overflow occurs while the application is executing in kernel land, and thus is using its kernel stack.

As you probably recall, the kernel mode stack is simply a small kernel memory block allocated from the physical page allocator just like any other memory-based resource. Compared to the user stack, it is generally quite small, it cannot grow on demand, and its state is discarded each time the kernel hands control back to the user-land process. This does not mean the kernel stack is reallocated each time, however. It simply means the stack pointer is moved back to the start each time the kernel is entered on behalf of the process.

By far, the most common example of stack corruption is the *stack overflow*, as shown in Figure 3.9.

There are three main approaches to exploiting a kernel stack corruption: overwrite the *return address*, overwrite some *local variable*, and overwrite the *adjacent page*. On some combination of operating systems and architectures (e.g., Linux on x86),

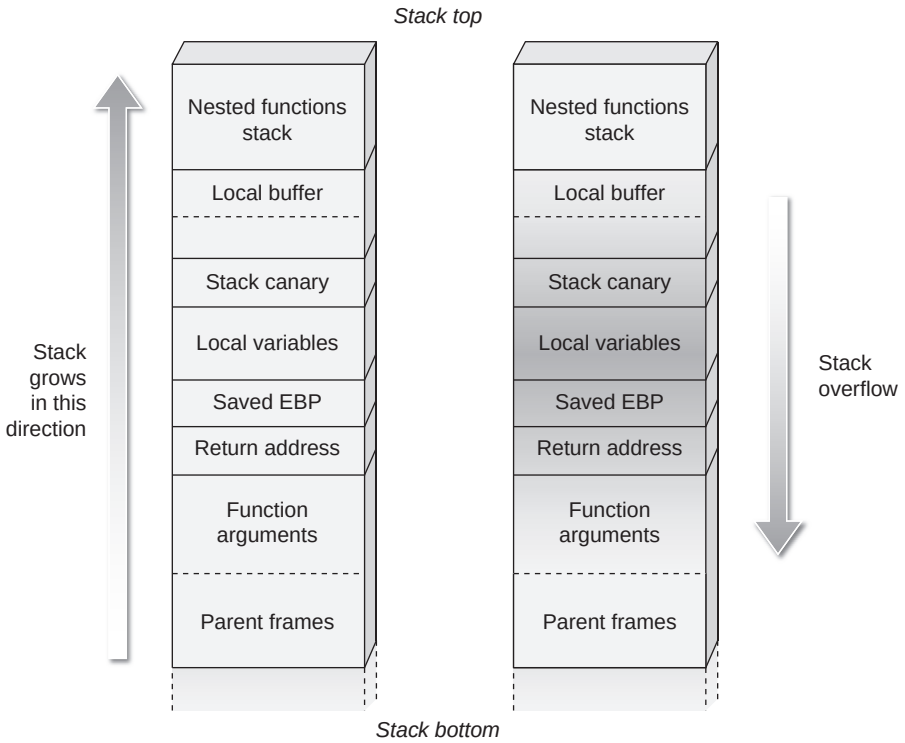

FIGURE 3.9

Stack overflow.

the same pages used to hold the stack are used to keep, at the end of the allocated pages, a controlling structure for the running process. This makes it easy to identify the current running process via a simple *AND* operation with the stack pointer value. Since such a structure is positioned at the bottom of the pages used for the stack, an overflow such as the one in Figure 3.9 cannot reach it (a write happens on increasing, not decreasing, addresses). Theoretically speaking, though, another problem might arise: a sufficiently long, nested sequence of calls could reach the bottom of the stack. Although such a vulnerability has never been found in any kernel (kernel developers are pretty careful about how they use the stack, and interrupts nowadays usually have an architecture-supported or software-provided alternate stack), we mention it here for completeness.

Overwriting the Return Address

Stack overflow exploitation based on overwriting the saved return address to hijack the control flow has been used successfully for more than two decades and is still fashionable. As an example, the advanced *return into kernel text* technique

that we discussed in the section "The Execution Step" is based on overwriting the saved instruction pointer.

Usually, to reach the saved return address you overflow a bunch of other local variables. If any of these variables is used before the function returns, you need to emulate its value, that is, set it to a value that will let the function get out correctly. As an example, if the function before exiting attempts to read from a pointer saved on the stack, you must be sure that you overwrite its value with an address of a readable memory area in the kernel. After the (eventual) local variable recovery, it is just a matter of applying the techniques we already described.

In an attempt to prevent canonical stack buffer overflows, a protection known as a *stack canary* has been designed and implemented inside compilers. The idea is pretty simple: A pseudorandom value, the *canary*, is pushed right after the return address and is checked when the called procedure returns. If the resultant value differs from the original value, that's a sign of a stack overflow. Activating stack canary protection is usually just a matter of turning on a compiler option and adding some handling code to be triggered whenever an overflow is detected. The easiest thing such handling code can do is to simply print some error message and panic the machine (a panic is safer than a compromise). Usually, to reduce the impact on performance, the compiler selects functions that are considered "potentially dangerous" and "patches" only those. An example of such a function could be one with at least some amount of space used on the stack.

A stack canary is a good protection scheme, but it suffers from a few problems:

- A particularly controlled overflow (e.g., an index-based overflow on an array saved on the stack) can write *past* the canary without touching it.
- The canary needs to be saved somewhere in memory, and thus can be revealed by a memory leak. In today's implementations, it is common to have a per-process stack canary, which basically gets computed at process creation and used (eventually with some permutation based on the state of some register) for the lifetime of the process. That means that once the canary is leaked one time in a function call inside a kernel path, subsequent calls by the same process going through the same path will have the same canary value at the specific function call.
- The canary cannot protect against the overflow of local variables placed before the canary itself.
- On an SMP system, you might be able to overflow to an adjacent page and get its code executed before the stack canary check is done. If enough recovery is performed by the shellcode, the canary could be restored before the check.

Note that despite becoming increasingly popular at the time of this writing stack canary protections are still not common (or turned on by default) on many operating systems.

Overwriting a Local Variable

Among the options we listed to bypass stack canary protection, we mentioned the possibility of *overwriting a local variable*. In fact, on various occasions, that may turn out to be easier than a classic overwriting of the saved return address. You trash only stack space that is local to the function, and you do not need to perform any general recovery of the stack state to safely return from the function.

The idea behind this technique is to find some sensible variable on the stack and turn the stack overflow into another type of vulnerability. Common situations include (but are not limited to):

- Overwriting a stored function pointer (e.g., inside a local static allocated structure)
- Overwriting a pointer later used in a copy operation, therefore turning the vulnerability into an arbitrary read or an arbitrary write (depending on how the pointer is used)
- Overwriting a stored (maybe precomputed) integer value, generating an integer issue

Race Conditions

Shared resources in kernel land are literally everywhere. Each kernel control path needs to correctly acquire and release whatever type of lock protects the shared resources it needs.

> **NOTE**
> We already briefly discussed locks during the analysis of the recovery step in the section "The Execution Step," so we won't discuss them again here.

A failure in correctly releasing a lock may make the associated resource unusable forever or, worse, trip on some kernel check and panic the machine or drive the kernel into a deadlock state (a situation where all the processes are stuck because each one depends on the resources that another one acquired). A failure in correctly acquiring a lock can lead to various corruptions and vulnerabilities, because the kernel task currently holding the lock expects and relies on the resources it locked down to not change. A similar situation occurs when a locking mechanism is not designed correctly. A classic example is leaving an opened window between when a process is picked up from the process list and when its privileges are changed. For a small window of time, an attacker could be able to manipulate (e.g., attach for debugging) a process that is about to become privileged (and thus unattachable for debugging by the attacker). It is worth mentioning that misuse of the locking mechanism is not the only source of race condition; a classic example is given by some *TOCTOU* (*time of check, time of use*) vulnerabilities involving the validation and subsequent access of user-land

data. In such issues, a kernel path loads and validates some value from user land, and then slightly afterward loads it again and uses it without revalidating. We will provide examples of successful exploits against this kind of vulnerability in Chapters 4 and 6.

Race conditions can be generated either by multiple kernel control paths running concurrently on different CPUs (as is the case on an SMP system) or by different paths running interleaved on a single CPU. Race conditions are always exploitable on SMP systems; however, sometimes the window might be very small and the race may be hard to win, resulting in only a subset of race conditions being exploitable on UP systems. The key point in each race is to increase your odds of winning. This is the topic of this section.

Kernel Preemption and the Scheduler

In Chapter 1, we introduced the scheduler and described it as the entity that moves the various tasks competing for execution into and out of the CPU. Since the goal of race conditions is basically to execute before the window closes, it is of utmost importance to understand the interaction between user/kernel tasks and the scheduler. A given path gets scheduled off the CPU in two circumstances:

- It voluntarily relinquishes the CPU, directly calling the scheduler. This is the case, for example, with some blocking locks. The process tries to acquire it but the lock is not available, so instead of spinning, it puts itself to sleep and invokes the scheduler to pick up another process. A similar situation occurs when waiting for a specific resource to be available; for example, for some I/O to complete and bring in a desired page of memory from disk.
- It is evicted from the CPU by the scheduler; for example, when the task-associated time frame or CPU quantum has expired. This is routine behavior for the scheduler, and it's how the operating system achieves multitasking and good responsiveness in the eyes of the user. If a kernel path can be interrupted during its execution to give the CPU to some other process, we define the kernel as *preemptable*.

At this point, a new task/process gets picked up and a new CPU quantum is given to it. Understanding what process will be picked next is as important, from a race exploitation point of view, as managing to make the scheduler execute and select a new process to run.

The scheduler uses different metrics to select the process to execute next, and some of them can be influenced directly from user land. Operating systems usually assign a priority to each process when it is created. The scheduler may take this priority into consideration when it selects the next CPU consumer. A process usually needs higher privileges to be able to raise its own priority, but it is always allowed to lower it. On a low load environment (an environment where not many CPU-intensive processes are active at the same time), lowering the priority at the right time might be enough to influence some scheduler decision and allow you to exploit the race window. This is especially important if you

are trying to exploit the race on a UP system, since relying on the scheduler to properly interleave your processes is the only way to generate the issue in the first place.

On SMP systems, you have one more shot (which theoretically makes any race condition exploitable). It is based on *binding* different processes to different CPUs (an operation always allowed on unprivileged tasks) and synchronizing their execution through the use of high-precision timers. Binding a process to a CPU means the process will compete to execute only on the specific CPU, and will remove it from competition on any other CPU. This is useful to prevent processes from interfering with each other on scheduling decisions.

There are multiple ways to ask the kernel for timing information, but since we need high precision, we cannot afford to incur any added kernel overhead. So, once again we exploit the architecture. Keeping with the convention of this book, we'll show an example of how to use the x86-64 architecture.

The x86-64 architecture provides access to an internal timer, the *TSC* (time stamp counter), which is a 64-bit machine-specific register that is set to zero at each reset of the machine and is updated at each clock cycle. Unprivileged user-land processes can query the value of this register by means of the *RDTSC* (Read *TSC*) instruction, which copies the 32 most significant bits of the *TSC* register into the *EDX* register and the 32 lowest significant bits into the *EAX* register. This approach is an excellent way to gather high-resolution timing information without incurring much overhead in execution time.

NOTE

The operating system can inhibit the *RDTSC* instruction by setting the *TSD* flag (Time Stamp Disable) in *CR4* (Control Register #4). Since the *TSC* is exploited by user-land applications, at the time of this writing this is not done by any operating system.

Exploitation Techniques

There are three main subsets of kernel race exploitation techniques, depending on the characteristics of the critical section you are targeting. We'll present the scenarios in order of complexity, which means that a technique that works successfully in the first one will definitely also work in the second one (and so on). Usually, though, the following techniques are based on a few more valid assumptions relative to the specific scenario, and are thus more effective and reliable.

The Critical Section Cannot Reschedule

In such a situation, the scheduler will not be called during execution of the critical section. This is usually the case when the race condition issue afflicts a deferred function or an interrupt/exception handler. In such situations, the kernel control path may not be able to reschedule for different reasons: it has already acquired a lock, it is running in interrupt context (and thus there is no backing process to put

to sleep to relinquish the CPU), or preemption has been temporarily disabled, for instance. This kind of race is the hardest to exploit, and since there is no scheduler involved, it is exploitable only on SMP systems with the help of high-resolution timers. The parameters you carefully need to take into account when you decide on which timer delay values to synchronize the user-land processes are the CPU frequency and the average time needed to reach the two racy critical sections. If the exploit is designed properly, it could keep on trying until the window is exploited. This is usually easier with race conditions because until the perfect conditions are met the kernel state is not affected.

The Critical Section Can Reschedule but Does Not Access User Land

This is probably the most common scenario with respect to kernel race conditions generated during a system call kernel path. Such issues are generally exploitable on UP systems, too, but an SMP system puts the odds more in our favor. A key point regarding these vulnerabilities concerns how the scheduler is involved. If you can drive the path into *voluntarily* relinquishing the CPU you have a much better shot at exploiting the vulnerability. This case usually leads to some blocking function that you can influence. For example, a memory allocation routine may block if no memory is currently available. By requesting and actively using a lot of memory with a user-land application you can generate such a situation.

If you instead need to rely on the scheduler to evict the current running process, this vulnerability becomes exploitable on UP only on a preemptable kernel. Preemptable kernels are the trend today, and schedulers are getting increasingly fair toward user-land processes. The catch here is to manage to get to the critical section with the kernel path that has basically finished its CPU time quantum, and have a CPU-intensive user-land application ready to demand the CPU to generate the race. Again, high-precision timers have a determinant role in correctly synchronizing the various threads/processes. On an SMP system, the exploitation of these issues is a lot easier, and is just a matter of having an acceptable measurement to synchronize the execution of the two (or more) threads.

The Critical Section Accesses the User Space

This is by far the easiest type of race to exploit. Since the kernel path accesses user land, you can play a trick to force it to sleep and thereby increase the size of the exploit window. Whenever you are accessing a user-land buffer, even a kernel implementing a combined user/address space model cannot simply dereference it. First, it needs to check that the address is below the split limit address. Second, it needs to ensure that the user-land mapping is valid so that the machine does not panic while attempting to reach it. Moreover, the kernel needs to be ready to react if the address is effectively part of the user address space, but the pages that back it are still on or have been swapped to disk. For example, a process may ask the kernel to map a file into memory. In such a situation, the kernel will create a valid mapping as large as the file is, but it will not allocate physical memory pages with the contents of the file. If, and only if, the process attempts to read one of them

will the kernel react to the fault and bring in the desired page from disk. This process is at the heart of the demand paging approach we mentioned in Chapter 1.

This specific operating system property gives us a pretty good weapon to exploit this type of race condition. In fact we can:

1. Map a file into memory or map a large portion of anonymous memory.
2. Place our kernel buffer on the boundary between two pages—one page that we ensure is mapped in and one that we are forced to page out.
3. Make the kernel path access the buffer on the boundary and go to sleep while the page fault handler code brings in the second page.
4. Get our thread scheduled and generate the race.

We mentioned forcing the second page out of memory. You can do this by digging into the operating system page cache implementation. Usually, this means you must predict how many pages will be paged in after an access (the operating system tries to exploit the principle of locality and brings in more pages, trying to avoid future slow calls to the page fault handler), or force the pages to be swapped to disk (e.g., generating a lot of the activity to fill the page cache), or a combination of the two.

We will provide some practical examples of this kind of attack in Chapters 4, 5, and 6.

THE INFORMATION-GATHERING STEP

The information-gathering step refers to all those pre-exploitation operations that our code will perform to collect information about and from the environment. During this phase, it is important to heed the following:

- **Do not panic the target** This is the kernel exploitation dogma. The information-gathering step allows you to decide at runtime if you should continue with the exploitation step. As an example, imagine that your exploit trashes a kernel structure and then forces a dereference of the corrupted function pointer. On an untested kernel version, the relative position of this pointer may have changed. In such a case, your exploit should detect the situation and give you a chance to stop so that you have time to check the specific version and come back later with a working version. As a general rule, it is better to fail than to panic a target. A panicked target is a lost target (the machine is down and far too much noise has been generated on the target box).
- **Simplify the exploitation process** In other words, use any information the system provides to obtain a better and safer entry point for your shellcode. Say that you have an arbitrary write at the kernel level. You could attempt to write to some address that seemed to be reliable on your tests. But how much better would it be if the system could tell you where to write? And if the system does not cooperate (say, in the presence of some kernel protection), how cool would it be if the underlying architecture could tell you?

These two advantages are obviously tightly linked. The second one allows you to write one-shot exploits that work on a large variety of targets, and thus reduce the odds of panicking a machine. It is important, though, to always attempt to validate the information you gather as much as possible. For example, say you have an arbitrary write issue and you are able to infer a destination address. In a combined user/kernel address space environment, you should at least check this value against the user/kernel-land split address. Moreover, if you are expecting this address to be in a particular area of the kernel, you may want to check it against known layout information (in Chapters 4, 5, and 6, we will provide detailed descriptions of typical kernel layout/addresses).

So far, we mentioned information that is *provided* from the environment. It does not depend on a vulnerability on the kernel, but simply on the clever use of the architecture and its interfaces. However, there is one more potential source of information, which is the consequence of *infoleaking bugs*. The classic infoleak bug is an arbitrary read at the kernel level. You can read portions of kernel memory from user land. In general, an infoleak simply pushes out to user land information that should not be exposed. As another example, think of a structure allocated on the stack, initialized on some of its members, and then copied back to user land. In such a case, the dead stack under the noninitialized member is leaked back to user land. Such issues are usually quite underrated, since in many cases they cannot lead to a direct exploitation. Unfortunately, this is a pretty bad habit: especially on systems with advanced kernel-level protections, a simple infoleak might give an attacker the missing piece of a one-shot reliable exploitation puzzle.

NOTE

Since local kernel exploits are far more common than remote ones, the remainder of this chapter focuses mainly on local information gathering. We will cover remote information gathering together with remote exploitation techniques in Chapter 7.

What the Environment Tells Us

Let's start our analysis of information-gathering approaches with what the environment we sit in tells us. Even operating systems with some level of hardening expose a good deal of information back to user land. Some of this is mandatory for correct execution of legitimate user-land applications (know where the kernel split address is or what version of the operating system is running); some of it is useful to give the user a chance to debug a problem (list if the specific module is loaded, show the resource usage of the machine); some of it is exposed by the architecture (as we mentioned in the *TSC/RDTSC* example we provided earlier when discussing race conditions); and a lot of it is simply underrated, and thus

weakly protected (the number of heap objects allocated in the kernel, the list of kernel symbols).

It is really interesting to see how just a few pieces of seemingly unconnected or useless information can be leveraged to sensibly raise the odds of a successful and reliable exploitation.

What the Operating System Is Telling You

The first piece of information we can easily grab from the system is the exact version of the running kernel. The kernel is a continuously evolving piece of software, and during an exploit we are likely to target a variety of its structures and interfaces. Some of them could be internal, and thus change from version to version, and some might have been introduced or dropped after a given release. This may require slightly different shellcodes or approaches between even minor releases of the same kernel. For example, the presence of a specific Windows Service Pack may drop an API tied with a vulnerable kernel path, or two different Linux kernel releases with just a minor version number mismatch may use a totally different internal credentialing structure. All operating systems offer an interface to user land to query the specific kernel version. We will discuss each one of them in Part II of this book.

Another interesting piece of information, especially on modular kernels, is what set of modules have been loaded and what (usually larger) set is available. Again, nearly all operating systems offer a way to query the kernel about its loaded modules, and usually return valuable pieces of information, such as the virtual address at which they have been loaded and their size. This information might come in handy if you are looking for specific offsets for an exploit. If this information is filtered (which is the case when extra security protections are in place) and your goal is only to detect if a specific module is available, you may be able to list (or even read) the available modules from the directory where they are kept. Moreover, nearly all modern operating systems implement a sort of automatic module loader to load a specific module only if the system really needs it. Thanks to this property, we can force the load of a vulnerable or useful module from user land by simply generating the right request.

Continuing our quest for information, on nearly all flavors of UNIX there is a program to print the kernel log buffer to the console: `dmesg`. Again, this buffer may contain valuable information, such as valid virtual address ranges or module debugging messages. For these reasons, Mac OS X "breaks" this UNIX tradition and prevents an unprivileged user from dumping the kernel control buffer and doing some security protection patches such as, for example, GRSecurity on Linux.

One of the most interesting types of information that we might be able to infer regards the layout of the kernel in memory and, especially, the addresses at which its critical structures or its *text* (the executable binary image) are mapped. One straightforward (and surprisingly effective) way to achieve this information is to look for the binary image of the kernel on disk. On many systems, administrators

forget to strip away unprivileged users' read permissions from that file (generally the default setting). Sometimes this is not even considered as having security implications! If you think back to our advanced *return into kernel text* technique, you can see how vital such information can be. Not only do we have access to all the symbol (function, variable, and section identifier) values/addresses, but also we can actually see the disassembly of each of them. In other words, we can deduce where a specific function or *opcode sequence* is in memory.

If the kernel binary image is not available (e.g., because it is on a boot partition that gets unmounted after boot time or the sysadmin has correctly changed its permissions), we can turn to the kernel-exported information. It is common, in fact, to have the kernel export to user land a list of its symbols through a pseudo-device or a file (as Linux does, for example, via */proc/kallsyms*). Again, by simply parsing this file we can discover the address of any structure or function at the kernel level. Let's see an example of how this file looks to better visualize the concept:

```
c084e7ad r __kstrtab_hrtimer_forward
c084e7bd r __kstrtab_ktime_get_ts
c084e7ca r __kstrtab_ktime_get_real
c084e7d9 r __kstrtab_ktime_get
c084e7e3 r __kstrtab_downgrade_write
c084e7f3 r __kstrtab_up_write
c084e7fc r __kstrtab_up_read
c084e804 r __kstrtab_down_write_trylock
c084e817 r __kstrtab_down_write
c084e822 r __kstrtab_down_read_trylock
c084e834 r __kstrtab_down_read
c084e83e r __kstrtab_srcu_batches_completed
c084e855 r __kstrtab_synchronize_srcu
c084e866 r __kstrtab_srcu_read_unlock
c084e877 r __kstrtab_srcu_read_lock
c084e886 r __kstrtab_cleanup_srcu_struct
```

As you can see, on the left of each symbol is its address. If this source is missing, we still have a way to try to figure out the kernel symbol layout, which is based on replicating the target environment somewhere else. This approach works pretty well with closed source operating systems such as Windows (by knowing the exact kernel version and the patches applied, it is possible to re-create an identical image) or with installations that are not supposed to manually update their kernels through recompilation. This second case is far more common than you may think for a lot of users. Recompiling either the Mac OS X or the Red Hat (Linux distribution) or the OpenSolaris kernel is just an extra burden (and would make the process of automatically patching and updating the system more complicated). Also, spotting what we can call a default kernel is extremely easy, thanks to the system version information we mentioned at the beginning of this chapter.

Kernel symbols, although dramatically useful, are not the only information we should hunt for, nor, unfortunately, the only information that will make an exploit reliable. In fact, they provide very good hints regarding the last stage of the triggering step (once we can divert execution to some address or we have an arbitrary write), but they help a lot less in the earlier stages, that is, when we are trying to generate the vulnerability.

We divided memory corruption vulnerabilities into two main families: heap and stack based. Also, we mentioned a common (last resort) technique for both of them, which is based on overwriting the adjacent page. In all those cases, to be successful we need to gather some information about how the various memory allocators work. Depending on the operating system, we may be able to get more or less detailed information. We will discuss the practical ways of doing this in Part II.

Once again, it is interesting to understand how we can leverage these seemingly harmless details in our exploit. Typical information that we might be able to gather about the heap allocator is the number of allocated and free objects for each cache. In the section "The Triggering Step," we said that our first objective when attacking the heap (or the physical page allocator) is to get to a state where allocator behavior is predictable. To do that, as we explained, we need to fill all the pages used for the cache (i.e., drive the allocation of all the free objects) so that the allocator will ask for new pages and start using them exactly as it was during its *very* first allocation. The kernel-exported information is of great importance, since it allows us to see how our indirect management of the allocator is going, and if any side effects are cropping up. By constantly monitoring the exported information, we can thus tune our exploit and, in most cases, turn it into a *one-shot* reliable exploit.

TOOLS & TRAPS...

Familiarize Yourself with Diagnostic Tools

The examples we have provided do not represent a complete list of all the information a system may expose; we just picked the ones that are most likely to be used in an exploit. It is usually worth it to spend some time becoming familiar with the unprivileged diagnostic tools that an operating system offers. Information such as the number and type of attached physical devices (e.g., PCI devices), the type and model of the CPU, or any kernel-exported statistic might come in handy in a future exploit. Operating systems tend to keep this information together—for example, providing a common interface to gather them up. We mentioned */proc/kallsyms* on the Linux kernel. On such a system, a tour of the */proc* (and */sys*) virtual filesystem will quickly give you an idea of the information you should be familiar with. We will go into more details about exploit-relevant exported information in Part II.

What the Architecture Tells Us

The architecture can be quite an ally, too. In general, two sources of information are particularly interesting in this regard: *counters* and *architecture-assisted*

software tables. The use of the high-precision time stamp counter (*RDTSC*/*TSC*) that we mentioned earlier is a good example of the former. In such a case, we obtain an incredibly accurate way to synchronize our attacking threads.

Architecture-assisted software tables are, to some extent, even more interesting. The idea behind such structures is pretty simple. There are some heavily used tables (e.g., the table that associates each interrupt to a specific handler) that are too expensive to implement purely in hardware. On the other hand, pure software support would greatly affect operating system performance. The solution to this issue is to have the software and hardware cooperate. The interrupt table is a good example of this. The architecture offers a register to keep track of the table's address and uses this information to internally and automatically perform the transition from a given interrupt number to the call of the specified handler. If each entry also contains other information (e.g., the privilege level required to call the specific routine), the architecture may or may not have support in place to deal with it in the hardware as well (e.g., the x86-64 architecture checks the *DPL* against the *CPL* and raises a fatal exception if the caller does not have enough privileges).

Obviously, the architecture needs to provide instructions to write and retrieve the address stored in the register holding the pointer to the software table. While the former is always a privileged operation, the latter is usually not.

In the section "The Execution Step" you saw how a crafted IDT entry can be the ideal way to reliably trigger your shellcode. Continuing the convention of focusing on the x86-64 architecture, take a look at the following code:

```
/* make IDT struct packed */
#pragma pack(push)
#pragma pack(1)
struct IDT
{
    USHORT limit;
    ULONG64 base;
};
#pragma pack(pop)

typedef struct IDT TYPE_IDT;

ULONG getIdt()
{
        TYPE_IDT idt;
        __asm {
                sidt idt
        }
        return idt.base;
}
```

When it is compiled in Microsoft Visual Studio C++ the preceding code will return the address of the IDT to an unprivileged process. The key point here is

the `__asm()` statement, which uses the *SIDT* (store interrupt descriptor table) instruction. This instruction copies the contents of the IDTR into the memory address specified by the destination operand. We just showed an example for the Windows platform, but what really matters here is to be able to execute an assembly instruction. Any compiler on any operating system gives us this possibility.

Once we know the address of the IDT we can calculate the correct offset from the start of the table to the interrupt handler that we want to hijack, and then apply the techniques described in the section "The Execution Step."

A similar approach applies to the GDT and the *SGDT* instruction. We will not go into the details here.

What the Environment Would Not Want to Tell Us: *Infoleaks*

As we mentioned earlier, there is a category of bugs that is usually a little underrated, and it is the one that leaks memory contents from the kernel. Unless the leak is pretty wide (you can retrieve a lot of kernel memory from user land) and/or very controllable (you can decide what area of the kernel to leak; note that in such a case you are usually able to leak as much memory as you want by repeating the attack), this kind of vulnerability does not lead to a compromise of the machine. These vulnerabilities are referred to as *information leaks* or *infoleaks*.

> **TIP**
> A large leak of kernel memory allows you to expose the contents of the physical pages currently in use by the system. Inside these pages you might find stored SSH keys, passwords, or mapped files that could lead to a direct compromise of the system.

This bug class is extremely useful in raising the efficiency of our exploit, especially if we are targeting a system configured with a lot of security protections (we will say a little more about that in the "Defend Yourself" sidebar at the end of this section), since it can cast a light on the addresses used in kernel land, and thus allow us to calculate the correct return address for our shellcode.

Leaks can occur on virtually any memory allocation, and thus can return information about:

- **Stack addresses/values** This is by far the most useful type of leak (after a full kernel memory leak, obviously), because you may not have any other way to deduce where your kernel stack is in memory. Also, a sufficiently controlled infoleak may reveal the presence of a canary protection and expose its value (allowing you to easily bypass that protection). Stack infoleaks become even more interesting when you consider that the kernel stack is generally not randomized. Since the kernel stack is allocated once and forever for a process, calling the same kernel path multiple times will lead to the same stack layout each time. An infoleak in such a situation could give you a precise offset to overwrite a pointer stored somewhere there.

- **Heap addresses/values** The generic case here is the ability to leak memory around an object, either before or after, or both before and after. Such a leak could expose information about the state of the previous/next object (if it is allocated or not), the type (say you have a general-purpose cache from which different types of objects are allocated), and its contents (for a free object, the value of the in-object control structures, if used, and for an allocated object, the values of its members, in case you need to replicate them during the overflow). Moreover, if the heap is protected with some form of randomized red zoning, the used check-value could be exposed and give you a way to bypass that protection, exactly as what happens with stack canaries.
- **Kernel data segment** The kernel data segment is the area created at compilation time that stores (global) kernel variables. An infoleak over this data could expose the value of some kernel configuration (is the specific protection active or not?) or, if you are not able to retrieve kernel symbols otherwise, give you a precise offset to use inside your exploit.

Today it is pretty common (and it is the ongoing trend) to have memory areas mapped as nonexecutable. If you are targeting a system that does not have this protection (e.g., a 32-bit x86 environment), a leak inside a memory area could also show interesting sequences of bytes that could be used as part of your shellcode (you should recall such an approach from the *return into kernel text* technique). Obviously, this is also the advantage that a kernel text infoleak could give, along with the possibility of checking if the specific vulnerability is there or not. This is useful if you need to stay under the radar on the target machine. Instead of executing an attack against a patched kernel (which may leave traces of the attempt on the target), you can check if the vulnerability is there and decide to proceed or not with the attack accordingly.

DEFEND YOURSELF

Make the Attacker's Life Difficult

After reading this section, it should be clearer how much use an attacker can make of seemingly harmless information or information leaking vulnerabilities. Projects such as GRSecurity for the Linux kernel aim to limit as much as possible both the exploitation vectors and the amount of information that an attacker can retrieve. Examples of this are the filtering of potentially interesting kernel-exported information (do not expose the symbol table or the heap state information to users) and the countermeasures to restrict some types of attacks (since there is no way to prevent a user from doing an *SIDT* instruction, just place the IDT inside a nonwritable mapping). Always check what options your operating system gives to restrict permissions to diagnostic tools and exported information. Note that removing the tools is not a viable option, since they are based on kernel-exported interfaces that the attacker can easily consume with his or her own tools. Also, do not leave a readable kernel image (the attacker can easily extract symbols out of it) or readable modules (the attacker might be able to trigger their loading) lying around. Note that a readable (potentially compressed) kernel image is available on most default system installations. The general idea here should be to strip away any information that the user does not need, no matter how irrelevant it could appear to be.

SUMMARY

This chapter was pretty meaty, as we discussed the major building blocks of a kernel exploit. Actually, we started a little before the exploit itself, focusing on the architecture level: the physical layer on top of which operating systems (and exploits targeting them) run. Following the theoretical-then-practical approach that characterizes not only this chapter but also the entire book, we discussed the common ideas behind architecture design and how the x86 and x86-64 architectures implement them.

Understanding the architecture helps you at various stages during exploit development. The first obvious application is during development of a shellcode: a sequence of instructions to which you try to divert execution. Moreover, architectural constraints and features influence the way the kernel behaves (e.g., with respect to memory management), and thus determine what you can and cannot do inside your attacking code. The architecture can also be an ally at various levels, providing both good entry points for your shellcode and vital information to improve the reliability of your exploit.

Going one step up from the architecture level, we focused on the execution phase of an exploit, the operations that you try to perform once you have successfully managed to hijack the execution path. There are two key points here: raise your privileges (eventually breaking out from any jailing environment) and restore the kernel to a stable state (releasing any resource that the targeted path might have acquired).

To successfully start the execution phase, you need to generate the vulnerability, hijack the execution flow, and redirect it to your payload. This is the job of the triggering phase. Generating the vulnerability is, obviously, vulnerability-dependent. You saw techniques for both heap and stack memory corruption vulnerabilities and race conditions. Hijacking the execution flow may happen immediately, as a result of using a modified return address from the stack, or it may be triggered later on, as a result of modifying some kernel structure and then calling a path using it.

The success (and reliability) of the triggering phase is highly influenced by how much information you have been able to gather about your target. We referred to this preparatory phase as the information-gathering phase. First, operating systems export a variety of seemingly harmless information. Your goal is to combine the various pieces and use them to increase the reliability of your exploit. Information such as the kernel symbols, the number of available CPUs, the kernel addresses, and the loaded modules can all play a significant role in transforming proof-of-concept code into a one-shot exploit, especially when targeting hardened environments. On such systems, though, a lot of this information might be filtered. In such a case, you need to look for/rely on information-leaking vulnerabilities, or bugs that allow you to peek at a more or less vast amount of kernel memory.

Related Reading

Architecture Design

Hennessy, John, and Patterson, David. 2003. *Computer Architecture—A Quantitative Approach* (Morgan Kaufmann).

Tanenbaum, Andrew, S. 2005. *Structured Computer Organization* (Fifth Edition) (Prentice-Hall, Inc.).

X86/x86-64 Architecture Manuals

Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 1: Basic Architecture (www.intel.com/products/processor/manuals/).

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference (www.intel.com/products/processor/manuals/).

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide (www.intel.com/products/processor/manuals/).

Exploiting Techniques

Advanced return-into-lib(c) exploits; www.phrack.orghttp://www.phrack.com/issues.html?issue=58&id=4/issues.html?issue=58&id=4.

Koziol, Jack, Litchfield, David, Aitel, Dave, *et al.* 2004. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes* (Wiley).

Krahmer, Sebastian. "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique"; www.suse.de/~krahmer/no-nx.pdf.