Return to Libc - Return Oriented Programming
Deadine: Midnight 7/5/2017

1 Lab Tasks

1.1 Initial Setup

You need to setup an `Ubuntu` x86_64 virtual machine. It is common practice for security researchers to study malicious code and infected systems under a virtualized environment in order to isolate the effects and keep the host system safe. You can use any virtualization software you like, depending on the host system. Most common choices include VMWare and VirtualBox, both available for Windows, Linux, Mac OS X. Another Linux-only choice is QEMU/KVM. The latest official `Ubuntu` installation image is available for download at: `http://releases.ubuntu.com/16.04/ubuntu-16.04.2-desktop-amd64.iso`. Linux distributions have implemented several security mechanisms to make attacks difficult. To simplify our job, we need to disable some of them first.

Address Space Randomization. `Ubuntu` Linux and several other operating systems use address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps in certain attacks. In this lab, we disable these features using the following commands:

```
$ su root
  Password: (enter root password)
#sysctl -w kernel.randomize_va_space=0
```

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. In the presence of this protection, some stack manipulations will not work. You can disable this protection if you compile the program using the *-fno-stack-protector* switch. For example, to compile example.c with Stack Guard disabled, you may use the following command:

```
$ gcc -fno-stack-protector - example example.c
```

Non-Executable Stack. `Ubuntu` used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel and dynamic linker use this marking to decide whether to make the stack of this running program executable or non-executable. In recent versions of `gcc` the stack is set to be non-executable by default. To change executable / non-executable stack behavior, use the following option when compiling programs:

```
For executable stack:
$ gcc -z execstack  -o test test.c

For non-executable stack:
$ gcc -z noexecstack  -o test test.c
```

As the objective of this lab is to show that the non-executable stack protection, although good, is not enough you should always compile your program using the `"-z noexecstack"` option in this lab.

1.2 The Vulnerable Program

```
/* retlib.c */

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

int bof(FILE *badfile) {
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 64, badfile);

    return 1;
}

int main(int argc, char **argv) {
    FILE *badfile;

    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");

    fclose(badfile);
    return 1;
}
```

Compile the above vulnerable program and make it set-root-uid. You can achieve this by compiling it in the `root` account, and `chmod` the executable to `4755`:

```
$ su root
  Password (enter root password)
# gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
# chmod 4755 retlib
# exit
```

The above program has a buffer overflow vulnerability. It first reads an input of size 40 bytes from a file called "badfile" into a buffer of size 12, causing the overflow. The function fread() does not check boundaries, so buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called "badfile". This file is under users' control. Now, our objective is to create the contents for "badfile", such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

1.3 Task 1: Exploiting the Vulnerability

Create the **badfile**. You may use the following framework to create one.

```
/* exploit.c */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z etc. */
  *(long *) &buf[X] = some address ;
  *(long *) &buf[Y] = some address ;
  *(long *) &buf[Z] = some address ;

  ...

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

You need to figure out the values for those addresses, as well as to find out where to store those addresses. If you incorrectly calculate the locations, your attack will probably not work.

After you finish the above program, compile and run it; this will generate the contents for "badfile". Run the vulnerable program `retlib`. If your exploit is implemented correctly, when the function `bof` returns, it will return to the `system()` libc function, and execute `system("/bin/sh")`. If the vulnerable program is running with root privilege, you can get the root shell at this point.

It should be noted that the `exit()` function is not really necessary for this attack; however, without this function, when `system()` returns, the program might crash, causing suspicions.

```
$ gcc -o exploit exploit.c
$./exploit          // create the badfile
$./retlib           // launch the attack by running the vulnerable program
# <---- You've got a root shell!
```

Questions. In your report, please answer the following questions:

- Please describe how you decide the address values. Either show us your reasoning, or if you use trial-and-error approach, show your trials.
- After your attack is successful, change the file name of `retlib` to a different name, making sure that the length of the file names are different. For example, you can change it to `newretlib`. Repeat the attack (without changing the content of `badfile`). Is your attack successful or not? If it does not succeed, explain why.

1.4 Task 2: Address Randomization

For this task, turn on the Ubuntu address randomization protection. Run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your return-to-libc attack difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on address randomization:

```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=2
```

1.5 Task 3: Stack Guard Protection

For this task, turn on Stack Guard protection. Please remember to turn off the address randomization protection. Run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the Stack Guard protection make your return-to-libc attack difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to compile your program with the Stack Guard protection turned on.

```
$ su root
  Password (enter root password)
# gcc -z noexecstack  -o retlib retlib.c
# chmod 4755 retlib
# exit
```

2 Tools

2.1 PEDA - Python Exploit Development Assistance for GDB

Peda is an add-on to GDB. It enhances its display: colorizes and displays disassembly codes, registers and memory information during debugging. At the same time, it adds commands to support debugging and exploit development.

You can download peda and find installation instructions at `https://github.com/longld/peda`.

The most important commants added by PEDA are the following (for a full list of commands use peda `help`):

- aslr – Show/set ASLR setting of GDB
- checksec – Check for various security options of binary
- dumpargs – Display arguments passed to a function when stopped at a call instruction
- dumprop – Dump all ROP gadgets in specific memory range
- elfheader – Get headers information from debugged ELF file
- elfsymbol – Get non-debugging symbol information from an ELF file
- find – Look for a symbol / string within a memory range (or within all memory)
- lookup – Search for all addresses/references to addresses which belong to a memory range

- patch – Patch memory start at an address with string/hexstring/int
- pattern – Generate, search, or write a cyclic pattern to memory
- procinfo – Display various info from /proc/pid/
- pshow – Show various PEDA options and other settings
- pset – Set various PEDA options and other settings
- readelf – Get headers information from an ELF file
- ropgadget – Get common ROP gadgets of binary or library
- ropsearch – Search for ROP gadgets in memory
- searchmem—find – Search for a pattern in memory; support regex search
- shellcode – Generate or download common shellcodes.
- skeleton – Generate python exploit code template
- vmmap – Get virtual mapping address ranges of section(s) in debugged process
- xormem – XOR a memory region with a key

For example, you can use the peda command `find` in order to look for the string "\bin\sh" in memory.

2.2 Ropper

You can use ropper to display information about files in different file formats and you can find gadgets to build rop chains for different architectures (x86/x86_64, ARM/ARM64, MIPS, PowerPC). For disassembly ropper uses the Capstone Framework.

You can download ropper and find installation instructions at `https://github.com/sashs/Ropper`. In the same page you will find use examples of ropper. Focus particularly on examples demonstrating the use of the `--search` argument.

3 Hints and Guidelines

3.1 Find out the addresses of libc functions

To find out the address of any libc function, you can use the following `gdb` commands (`a.out` is an arbitrary program):

```
$ gdb a.out

(gdb) break main
(gdb) run
(gdb) print system
 $1 = {<text variable, no debug info>} 0x9b4550 <system>
(gdb) print exit
 $2 = {<text variable, no debug info>} 0x9a9b70 <exit>
```

The actual addresses in your system might be different from these numbers.

3.2 Finding (or Putting) the shell string in the memory

One of the challenges in this lab is to find the string `"/bin/sh"` into the memory, and get its address. See Section 2 on tools to help you with that.

Alternatively, if you are not lucky, you can put it in memory yourselves, using environment variables. When a C program is executed, it inherits all the environment variables from the shell that executes it. The environment variable **SHELL** points directly to `/bin/bash` and is needed by other programs, so we introduce a new shell variable **MYSHELL** and make it point to our shell program (such as `sh`, `bash`, or `zsh`).

```
$ export MYSHELL=/bin/sh
```

We will use the address of this variable as an argument to `system()` call. The location of this variable in the memory can be found out easily using the following program:

```
void main(){
  char* shell =  getenv("MYSHELL");
  if (shell)
     printf("%x\n", (unsigned int)shell);
}
```

If the address randomization is turned off, you will find out that the same address is always printed out. However, when you run the vulnerable program `retlib`, the address of the environment variable might not be exactly the same as the one that you get by running the above program; the address can even change when you change the name of your program

3.3 Return Oriented Programming (ROP)

Unlike returning to a library and using existing code as it was (in some ways) intended, ROP takes advantage of unintended code sequences occurring in a program. The machine code of a binary is possible to be interpreted differently depending on the starting position of execution. Take for example, the following sequence of bytes within the executable code segment:

```
[addr]    [bytes]                [instr]
7ffc:     ...
8000:     4c 89 5f c3     mov    %r11,-0x3d(%rdi)
8004:     ...
```

If execution starts at address 0x800 (opcode '4c'), the instruction is the mov instruction that shows on the right. However, if the execution starts at address 0x8002 (opcodes '5f' and 'c3'), then the processor executes the following sequence of instructions:

```
[addr]    [bytes]                [instr]
7ffc:     ...
8000:     4c 89          ; skip the first two bytes (e.g. direct jump)
8002:     5f             pop %rdi
8003:     c3             ret
8004:     ...
```

ROP takes advantage of this behavior. Similarly to Return-to-Libc, the main idea is to chain together instructions, already present in parts of the memory marked as executable, using short instruction sequences ending in the 'ret' instruction. In ROP terminology these "hidden" sequences of instructions are called gadgets and can be used as fundamental elements for exploiting a buffer overflow. Note that gadgets may consist of concrete, unmodified instructions. For example, the epilogue of functions is usually a good place to find gadgets.

ROP gadgets allow the programmer to control the contents of the registers and memory in a reliable way. There are a lot of tools available on the web that can search inside an executable or library for useful gadgets. A common source of gadgets is the C dynamic library, as almost every program links with it. Some examples of gadgets in the libc.so dynamic library are outlined bellow:

```
0x0003aa08 : pop rax ; ret
0x00070bf8 : pop rbp ; jmp rax
0x0007ded2 : pop rbx ; xor edx, edx ; jmp rax
0x0001fa3f : cmp al, byte ptr [rax] ; pop rax ; pop rbx ; pop rbp ; ret
0x000f6618 : mov eax, 1 ; pop rbx ; ret
0x00104858 : pop r12 ; pop r13 ; pop r14 ; pop r15 ; jmp rax
```

ROP bypasses the stack execution protection mechanism in the same way the simple Return-to-Libc approach does. The objective for a succesfull ROP attack is to create a fake stack frame (return address, parameters) and modify the contents of specific registers in order to take control over the execution of a program. The ABI dictates the calling conventions i.e. the registers and memory locations used for a function call and return.

For example if the first two arguments of a fucntion are passed to the rdi and rsi registers (as specified in the x86 64-bit ABI), and we have found a gadget that modifies the values of these registers (e.g. *pop rdi; pop rsi; ret*), that means we can control the first two arguments of any function call! We can overflow the vulnerable buffer with such values that setup the appropriate registers and call the desirable function. A stack example after an overflow and ready to call the *int foo(int, int)* function looks like this:

```
                Stack
              ---------------
hi_addr:      ...
              foo()'s address [ fake frame ]
              arg2            [ fake frame ]
              arg1            [ fake frame ]
              return address  [ overflow: gadget's address]
              frame pointer   [ overflow ]
              ...             [ overflow ]
              ...             [ overflow ]
              buffer          [ overflow ]
low_addr:     ...
              ---------------
```

When the current function finishes, it will return to the gadget. Then the gadget assigns the arguments into the registers and returns to the foo() function. The fake frame in the stack simulates the gadget as a valid function call.

References

[1] c0ntext Bypassing non-executable-stack during exploitation using return-to-libc http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf

[2] Phrack by Nergal Advanced return-to-libc exploit(s) *Phrack 49*, Volume 0xb, Issue 0x3a. Available at http://www.phrack.org/archives/58/p58-0x04

[3] Saif El-Sherei Return-Oriented-Programming (ROP FTW) https://www.exploit-db.com/docs/28479.pdf