SE 421: Assignment #2

Due on September 7, 2018 at 12:00 PM (noon)

Instructor: Ben Holland

https://github.com/SE421/assignment2

Student Name:

Problem 1

(20 points)

A. (*0 points*) Lab Setup

A buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address. In this problem we will be exploiting a buffer overflow vulnerability in a small program Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them one by one, and see whether our attack can still be successful.

To complete the lab setup take a screenshot (or screenshots) of the following steps to document in your report.

A.1. First download the SEEDUbuntu16.04.zip file and start the contained virtual machine by following the setup instructions at `http://www.cis.syr.edu/~wedu/seed/lab_env.html`. The virtual machine credentials are: `seed:dees`.

A.2. Ubuntu and several other Linux-based systems uses address space randomization [1] to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks.

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

A.3. The GCC compiler implements a security mechanism called `StackGuard` to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. We can disable this protection during the compilation using the -fno-stack-protector option. For example, to compile a program example.c with StackGuard disabled, we can do the following:

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

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 or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of `gcc`, and by default, stacks are set to be non-executable (further reading: [3]). To change that, use the following options when compiling programs.

For executable stack:

```
$ gcc example.c -z execstack -o example
```

For non-executable stack:

```
$ gcc example.c -z noexecstack -o example
```

Our vulnerable program is `vulnerable.c` (`https://raw.githubusercontent.com/SE421/assignment2/master/vulnerable.c`). Compile `vulnerable.c` without stack protections (disable `StackGuard`) and with an executable stack. Name the compiled vulnerable executable program "`vulnerable`".

A.4. In both Ubuntu 12.04 and Ubuntu 16.04 VMs, the /bin/sh symbolic link points to the /bin/dash shell. However, the dash program in these two VMs have an important difference. The dash shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a Set-UID process. Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege. The dash program in Ubuntu 12.04 does not have this behavior.

Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in /bin/dash can be easily defeated). We have installed a shell program called zsh in our Ubuntu 16.04 VM. We use the following commands to link /bin/sh to zsh (there is no need to do these in Ubuntu 12.04):

```
$ sudo rm /bin/sh
```

```
$ sudo ln -s /bin/zsh /bin/sh
```

B. (*2 points*) Testing Shellcode

Before starting the attack, let us get familiar with the shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```c
#include <stdio.h>
int main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

The shellcode that we will use is just the assembly version of the above program. To test the shellcode download the test_shellcode.c program (https://raw.githubusercontent.com/SE421/assignment2/master/test_shellcode.c), compile it, and run it to see if a shell is invoked. Note that you should compile the test_shellcode.c program with the following command so that the stack is executable or else the program will fail.

```
$ gcc test_shellcode.c -z execstack -o test_shellcode
```

Change the owner of the test_shellcode program to root and set its permissions to 4755.

```
$ sudo chown root test_shellcode
```

```
$ sudo chmod 4755 test_shellcode
```

The shellcode invokes the execve() system call to execute /bin/sh. A few places in this shellcode are worth mentioning. First, the third instruction pushes "//sh", rather than "/sh" into the stack. This is because we need a 32-bit number here, and "/sh" has only 24 bits. Fortunately, "//" is equivalent to "/", so we can get away with a double slash symbol. Second, before calling the execve() system call, we need to store name[0] (the address of the string), name (the address of the array), and NULL to the %ebx, %ecx, and %edx registers, respectively. Line 5 stores name[0] to %ebx; Line 8 stores name to

%ecx; Line 9 sets %edx to zero. There are other ways to set %edx to zero (e.g., `xorl %edx, %edx`); the one (`cdq`) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the EAX register (which is 0 at this point) into every bit position in the EDX register, basically setting %edx to 0. Third, the system call `execve()` is called when we set %al to 11, and execute "`int $0x80`".

Include a screenshot of your working shellcode (you should get a root shell (# prompt) by running `test_shellcode`).

C. (*10 points*) Exploiting the Vulnerable Program

C.1. Change the owner of the vulnerable program to root and set its permissions to 4755.

```
$ sudo chown root vulnerable
```

```
$ sudo chmod 4755 vulnerable
```

Include a screenshot of the vulnerable program permissions by running:

```
$ ls -l
```

C.2. The provided `vulnerable.c` program has a buffer overflow vulnerability. It first reads an input from a file called `badfile`, and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` is only 24 bytes long. Because `strcpy()` does not check boundaries, 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 user's 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.

A partially completed C code to create the exploit file called `exploit.c` can be downloaded from `https://raw.githubusercontent.com/SE421/assignment2/master/exploit.c`.

After completing the commented TODOs, compile and run the *exploit.c* program to created the `badfile`. Then run the `vulnerable` program, which reads from the `badfile`. If you are successful you should get a root shell (# prompt).

```
$ gcc -o exploit exploit.c
```

```
$ ./exploit
```

```
$ ./stack
```

It should be noted that although you have obtained the "#" prompt, your real user id is still yourself (the effective user id is now root). You can check this by running "`id`" in your root shell.

```
# id
```

Hint: If you compile a program with debug symbols (add `-g` option to `gcc`), you can set a breakpoint inside the `bof()` function (`gdb$ break bof`), print the memory address of the start of the buffer (`gdb$ print buffer`), print the memory address of the `$ebp` register (`gdb$ print $ebp`), and even compute the distance between the two hex values (`gdb$ print (0x10 - 0x04)`). Using GDB you could learn the offset of where EIP is overwritten. Remember that 32 bit systems use

Problem 1 continued on next page. . .

4 bytes to represent an address.

Include a link to your completed `exploit.c` source, a short explanation of how you completed the exploit, and a screenshot of your successful exploitation of the `vulnerable` program.

D. (*2 points*) Defeating `dash's` Countermeasure

D.1. As we have explained before, the dash shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal to the real UID. This can be observed from dash program's changelog. We can see an additional check, which compares real and effective user/group IDs.

```
// https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.diff.gz
// main() function in main.c has following changes:

++ uid = getuid();
++ gid = getgid();

++ /*
++  * To limit bogus system(3) or popen(3) calls in setuid binaries,
++  * require -p flag to work in this situation.
++  */
++ if (!pflag && (uid != geteuid() || gid != getegid())) {
++    setuid(uid);
++    setgid(gid);
++    /* PS1 might need to be changed accordingly. */
++    choose_ps1();
++ }
```

The countermeasure implemented in `dash` can be defeated. One approach is not to invoke `/bin/sh` in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as `zsh` to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking `setuid(0)` before executing `execve()` in the shellcode. In this task, we will use this approach. We will first change the `/bin/sh` symbolic link, so it points back to `/bin/dash`:

```
$ sudo rm /bin/sh
```

```
$ sudo ln -s /bin/dash /bin/sh
```

D.2. To see how the countermeasure in dash works and how to defeat it using the system call `setuid(0)`, we write the following C program. We first comment out line 11 and run the program as a `Set-UID` program (the owner should be root); please describe your observations. We then uncomment Line 11 and run the program again.

```
1  // dash_shell_test.c
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5  int main()
6  {
7     char *argv[2];
8     argv[0] = "/bin/sh";
9     argv[1] = NULL;
10
11    // setuid(0);
```

```c
  execve("/bin/sh", argv, NULL);

  return 0;
}
```

The `dash_shell_test.c` program can be compiled and set up using the following commands (remember that we need to make it root-owned `Set-UID` program).

```
$ gcc dash_shell_test.c -o dash_shell_test
```

```
$ sudo chown root dash_shell_test
```

```
$ sudo chmod 4755 dash_shell_test
```

```
$ ./dash_shell_test
```

Describe your observations and include any necessary screenshots.

E. From our previous experiment, we will see that `setuid(0)` makes a difference. Let us add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke `execve()`. The updated shellcode adds 4 instructions: (1) set `ebx` to zero in Line 2, (2) set `eax` to `0xd5` via Line 1 and 3 (`0xd5` is `setuid()'s` system call number), and (3) execute the system call in Line 4. Using this shellcode, we can attempt the attack on the vulnerable program when `/bin/sh` is linked to `/bin/dash`.

```
char shellcode[] =
  "\x31\xc0" /* Line 1: xorl %eax,%eax */
  "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
  "\xb0\xd5" /* Line 3: movb $0xd5,%al */
  "\xcd\x80" /* Line 4: int $0x80 */
  // ---- the shellcode below is the same as the previous exploit shellcode ---
  "\x31\xc0"
  "\x50"
  "\x68""//sh"
  "\x68""/bin"
  "\x89\xe3"
  "\x50"
  "\x53"
  "\x89\xe1"
  "\x99"
  "\xb0\x0b"
  "\xcd\x80"
```

Using the modified shellcode above in `exploit.c`, try the attack again and see if you can get a root shell. Please describe and explain your results.

F. (*2 points*) Defeating Address Randomization

F.1. On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^19 = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. First, we turn on the Ubuntu's address randomization using the following command.

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

Rerun your exploit. Please describe and explain your observation.

F.2. We then use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the badfile can eventually be correct. You can use the following shell script to run the vulnerable program in an infinite loop. If your attack succeeds, the script will stop; otherwise, it will keep running. Please be patient, as this may take a while. Let it run overnight if needed. Please describe your observations.

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
   do
   value=$(( $value + 1 ))
   duration=$SECONDS
   min=$(($duration / 60))
   sec=$(($duration % 60))
   echo "$min minutes and $sec seconds elapsed."
   echo "The program has been running $value times so far."
  ./stack
done
```

G. (*2 points*) Investigate `StackGuard` Protection

Before working on this task, remember to turn off the address randomization first and non-executable stack protections, or you will not know which protection helps achieve the protection.

In GCC version 4.3.3 and above, `StackGuard` is enabled by default. Therefore, you have to disable `StackGuard` using the switch mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable `StackGuard`.

```
$ gcc vulnerable.c -o vulnerable -z execstack
```

For this task, you will recompile the vulnerable program, to use GCC StackGuard, execute the exploit again, and report your observations. You may report any error messages you observe.

H. (*2 points*) Investigate Non-executable Stack Protection

Before working on this task, remember to turn off the address randomization first and `StackGuard` protections, or you will not know which protection helps achieve the protection.

```
$ gcc vulnerable.c -o vulnerable -fno-stack-protector -z noexecstack
```

For this task, you will recompile the vulnerable program, to use non-excutable statck protections, execute the exploit again, and report your observations. You may report any error messages you observe. Answer the follow up question: How does the `return-to-libc` (return oriented programming) technique bypass this protection?

Problem 2

(Extra Credit) 10 points

Your goal is to gain access to the given victim virtual machine and exfiltrate the contents of a file named "flag.txt" that is located on the administrator user's desktop folder. Submit the contents of the flag.txt file as your solution along with a walkthrough (including screenshots) of how you successfully retrieved the flag. If you are unable to recover the flag.txt file contents, partial credit will be given for documenting your efforts up until the point you got stuck.

Victim VM: `http://www.benjaminsbox.com/pac/WinXPSP3Victim.ova`

Notes and Hints:

- The virtual machine was created and tested with VMware. You can download VMware Player for free for Windows/Linux at `https://www.vmware.com/go/tryplayer`. The free player only editions do not have snapshot abilities or VM creation abilities, however VMware offers ISU students free professional editions of VMWare Workstation (Windows/Linux) and VMWare Fusion (mac) at `https://cytools.iastate.edu/vmap`.
- After installing VMware Player or VMware Workstation or VMware Fusion, "import" the WinXPSP3Victim.ova file included with this assignment into VMware.
- You do not have the password to log into the virtual machine, so you will be unable to debug your exploit code. It may be helpful to mimic the target environment. The target virtual machine was constructed using the Windows XP SP3 ISO available at: (`http://www.benjaminsbox.com/pac/en_windows_xp_professional_with_service_pack_3_x86.iso`). When creating the virtual machine you do not need a license key (just run it in the 30 day evaluation mode). VMware Workstation and VMware Fusion both can be used to create a new VM.
- The Windows VM runs the MiniShare webserver as a *startup task* (see `https://support.microsoft.com/en-us/help/308569/how-to-schedule-tasks-in-windows-xp`). If you need to restart the MiniShare webserver then will need to reboot the virtual machine.
- The virtual machine is running the MiniShare 1.4.1 binary as Administrator.
- The virtual machine is configured to accept IP addresses via DHCP and was placed on a Host Only network (meaning only your host machine or another VM on your host can reach the Victim machine). From your host machine or attacker VM you should first confirm that you can reach the MiniShare web server with a web browser by navigating to `http://<VICTIM_IP>`. To find the victim IP address type "arp -a" in your *host machine* (machine running hosting the virtual machine) command line. Cross reference this IP list with the MAC address assigned by the VMware network adapter (shown under the VM network settings menu). As a last resort, if you are still having network connectivity issues, a TA can log into the VM for you to check network adapter settings.
- There are walkthroughs for exploiting MiniShare online as well as in the class lecture notes.
- Some commands in the tools provided by recent Kali Linux releases have changed slightly from when the class lecture notes were made. Note that there are premade releases of Kali virtual machines at: `https://www.offensive-security.com/kali-linux-vm-vmware-virtualbox-hyperv-image-download`.
- Use Google or come to the office hours if you get stuck.

Notice: Problem 1 of this assignment was adapted from the Syracuse SEED Buffer Overflow Vulnerability Lab developed under National Science Foundation under Award No. 1303306 and 1318814. The lab in its original form is available at `http://www.cis.syr.edu/~wedu/seed/Labs_16.04/Software/Buffer_Overflow`. By extension problem 1 of this assignment is licensed under a Creative Commons Attribution-NonCommercialShareAlike 4.0 International License. You are free to copy and redistribute the material in any medium or format. You must give appropriate credit. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You may not use the material for commercial purposes. You should temporarily disable ASLR with the command

References

[1] "Address space layout randomization," https://en.wikipedia.org/wiki/Address_space_layout_randomization, accessed: 2018-08-28.