How Does Buffer Overflow Attack Work

S. C. Kothari

CPRE 556: Lecture 7, January 31, 2006

Electrical and Computer Engineering Dept.
Iowa State University

Security: When is it software problem

- We can distinguish security problems by the mechanisms requiring changes to eliminate the vulnerability.
- Network Problem: requires changing networking mechanisms such as network protocols.
- OS Problem: requires changing OS mechanisms such OS resource management policies.
- Software Problem: requires changing software implementation or design

Security Bugs Can Be Expensive

- Buffer overflow in IIS
  - Estimated cost: $3.26 billion
- Buffer overflow in SQL Server
  - Estimated cost: $1.2 billion

What Entrances Do the Hackers Use

- Hackers exploit interactions with:
  - Operating System
  - User Interfaces
  - File System
  - Libraries

Buffer Overflow Attack (BOA)

- Deadly attack underlying many computer highjackings in the past.
- Dominate the area of remote network penetration vulnerabilities, where an anonymous Internet user seeks to gain partial or total control of a host.
- Present the attacker the ability to inject and execute attack code.
- Typically attack a root program and executes code similar to “exec(sh)” to get a root shell.
- The attack is possible with C and C++ programs, not with Java.

BOA Basic Idea

- The idea is simple: enter long strings into input fields, could be APIs/exposed internal objects
- This is an important bug because:
  - copy/paste into inputs fields is a fairly common practice
- Buffer overflow may be exploitable by a hacker to get arbitrary code to run on a system.

BOA Demonstration

- The demo developed with NSF support is available at: http://nsfsecurity.pr.erau.edu/bom/
- The demonstration illustrates the buffer overflow attack as a high-level concept.
- The demo makes simplifications and does not cover complex and subtle mechanism employed by BOA. These will be covered in this lecture.

BOA: One Possibility

- In this lecture we will discuss one possibility where the attacker corrupts the activation record (AR).
- Each time a function is called an AR is created on the stack.
- AR includes: return address, space for locals and arguments, an a pointer to the previous record.
- Buffer is a local variable.
- Attacker stuffs the buffer so that it overflows and replaces the return address with a new return address.
- Attacker stuffs code in the buffer and arranges the new return address to point to the code stuffed in the buffer.
- Called ‘Stack Smashing,’ - favorite of attackers (e.g. Morris Worm).

EX 1: Basics of Activation Record I

example1.c:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

The call to function() is translated to:

```
pushl $3
pushl $2
pushl $1
call function
```

Pushes the 3 arguments to function backwards into the stack, and calls function()

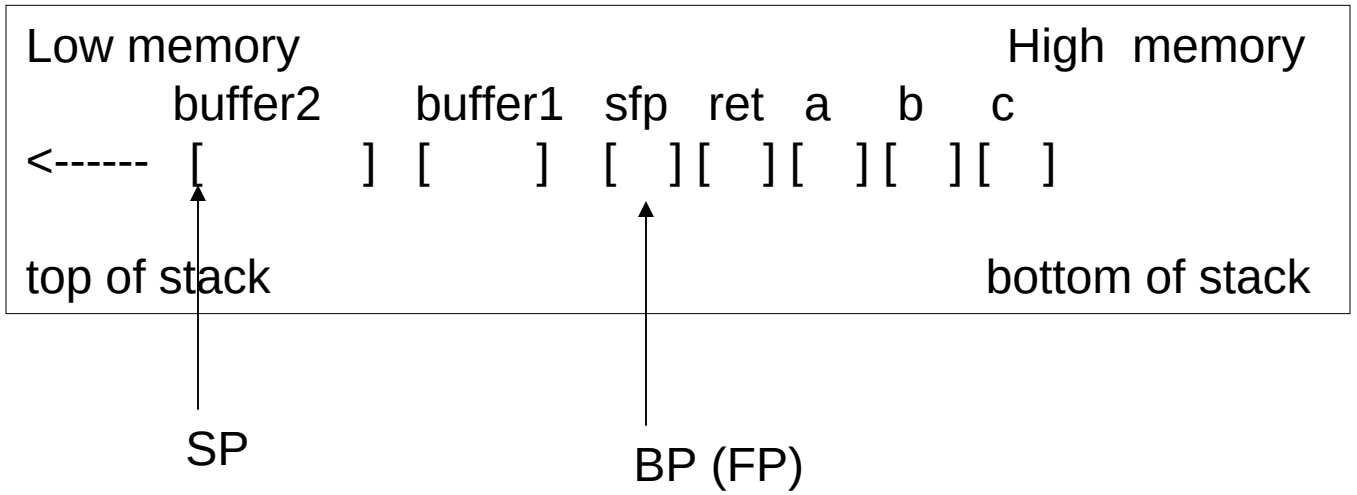
Prolog: The first thing done in function:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

Pushes EBP (frame pointer) onto the stack, copies the current SP onto EBP, then allocates space for the local variables by subtracting their size from SP.

Draw the stack

EX1: Basics of Activation Record II



Offsets relative to BP are used as addresses within the function to access local variables and parameters.

EX2: Buffer Overflow

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

large_string is filled with the character 'A'. It's hex character value is 0x41. That means that the return address is now 0x41414141. This is outside of the process address space. That is why when the function returns and tries to read the next instruction from that address you get a segmentation violation.

Example 3

example3.c:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 10;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

What is the outcome?

Example 3: What is the outcome?

example3.c:

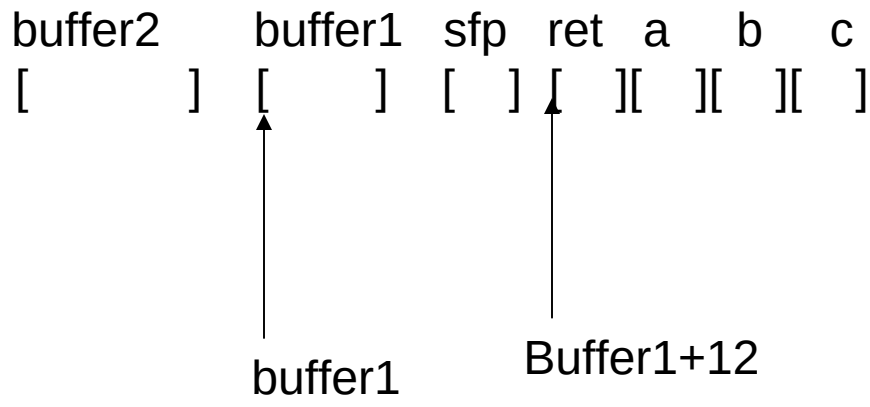
```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 10;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

The stack looks like:



So, we are incrementing the return address by 8.

To see what is at the new return address, let us see the dump of assembler code for function main generated using the gdb.

Example 3: What is the outcome?

example3.c:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 10;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

```
Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
0x80004c6 <main+54>:    ret
0x80004c7 <main+55>:    nop
```

+10 ↓

Note that we skipped the assignment x = 1.

How to make it execute other code?

- The attacker wants the program to spawn a shell. From the shell he can then issue other commands as he wishes.
- The code to execute is placed in the buffer and the return address is overwritten so it points back into the buffer.

BOA Tricks

- BOA involves many additional tricks.
- For example, one problem is we need to guess *exactly* where the address of our code will start. If we are off by one byte more or less we will just get a segmentation violation or a invalid instruction.
- Trick: Pad the front of our overflow buffer with NOP instructions. Fill half of the buffer with NOP instructions.
- Chances of successful attacks are increased: If the return address points anywhere in the string of NOPs, they will just get executed until they reach the code that the attacker really wants to execute.

References

- The examples are from the paper listed on the CPRE 556 website:
  - Smashing The Stack For Fun And Profit, Aleph One.