Buffer Overflow Attacks

Intermediate Level

How Serious of a Problem Is This?

- The effectiveness of the buffer overflow attack has been common knowledge in software circles since the 1980’s
- The Internet Worm used it in November 1988 to gain unauthorized access to many networks and systems nationwide
- Still used today by hacking tools to gain “root” access to otherwise protected computers
- The fix is a very simple change in the way we write array accesses; unfortunately, once code that has this vulnerability is deployed in the field, it is nearly impossible to stop a buffer overflow attack

Buffer Overflow Attacks: An Overview

- The buffer overflow attack exploits a common problem in many programs.
- In several high-level programming languages such as C, “boundary checking”, i.e. checking to see if the length of a variable you are copying is what you were expecting, is not done. This is shown in the following example (in C/C++):

```
void main(){
    char bufferA[256];
    myFunction(bufferA);
}
```

```
void myFunction(char *str) {
    char bufferB[16];
    strcpy(bufferB, str);
}
```

Buffer Overflow Attacks: An Overview

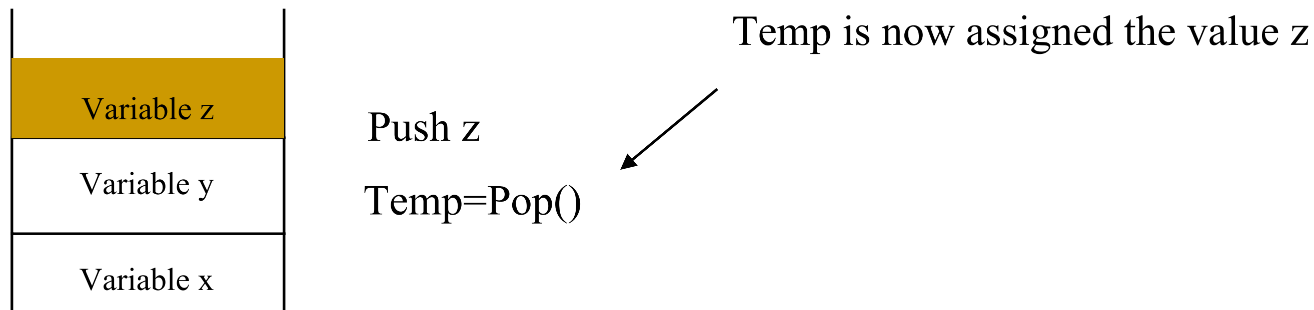
```
void main(){
    char bufferA[256];
    myFunction(bufferA);
}
```

```
void myFunction(char *str) {
    char bufferB[16];
    strcpy(bufferB, str);
}
```

- Main() passes a 256 byte array to myFunction(), and myFunction() copies it into a 16 byte array, attempting to fill bufferB[16] with 240 bytes of data. Since there is no check on whether bufferB is big enough, the extra data overwrites other unknown space in memory.
- This vulnerability is the basis of buffer overflow attacks. How is it used to harm a system? It modifies the system stack.

Stacks and Processes: An Overview

- Assume we are in main(), and we make a procedure call to myFunction(char *str).
- How does myFunction access the variables that are passed to it?
- How does the system know where to resume execution of main(), when myFunction() has terminated?
- This is done by creating a stack frame, a set of data on the program stack
- The stack works just like a regular stack data structure, with *push* and *pop.*



Stacks and Processes: An Overview

In our case, when we call myFunction(char *str), here is what happens.

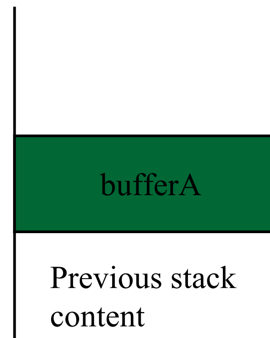
```
void main(){
    char bufferA[256];
    myFunction(bufferA);
}
```

When we get here, the OS executes the following instruction:

push(bufferA);

call myFunction;

Our stack now looks like this:

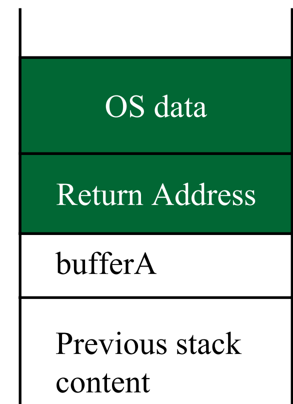


Stacks and Processes: An Overview

When “call myFunction” is executed, the return address of the main myFunction is also pushed into the stack. This address tells the OS what to execute after myFunction() is done.

In this case, it should return to the next line of code in main(). When this is done, other OS-related data is pushed into the stack.

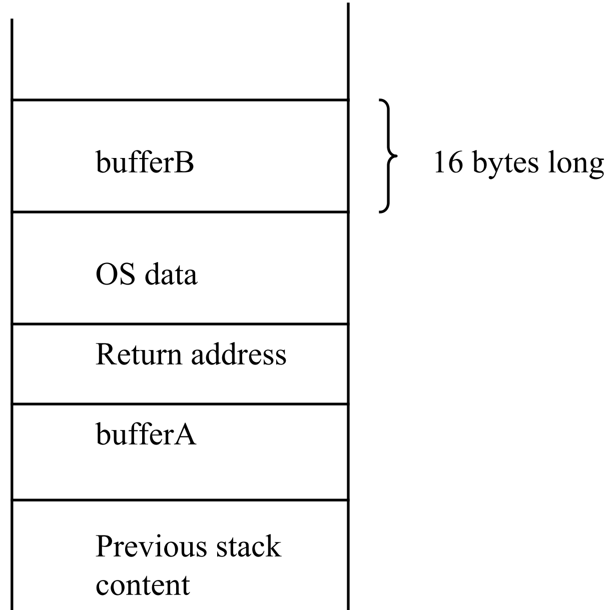
Now we are executing the code in myFunction(). As in any procedure, the first thing myFunction() does is push its local variables (bufferB[16] in this case) onto the stack. This variable, as defined in myFunction() is 16 bytes long. So the OS will allocate 16 bytes in the stack for it.



Stacks and Processes: An Overview

Our stack now looks like this:

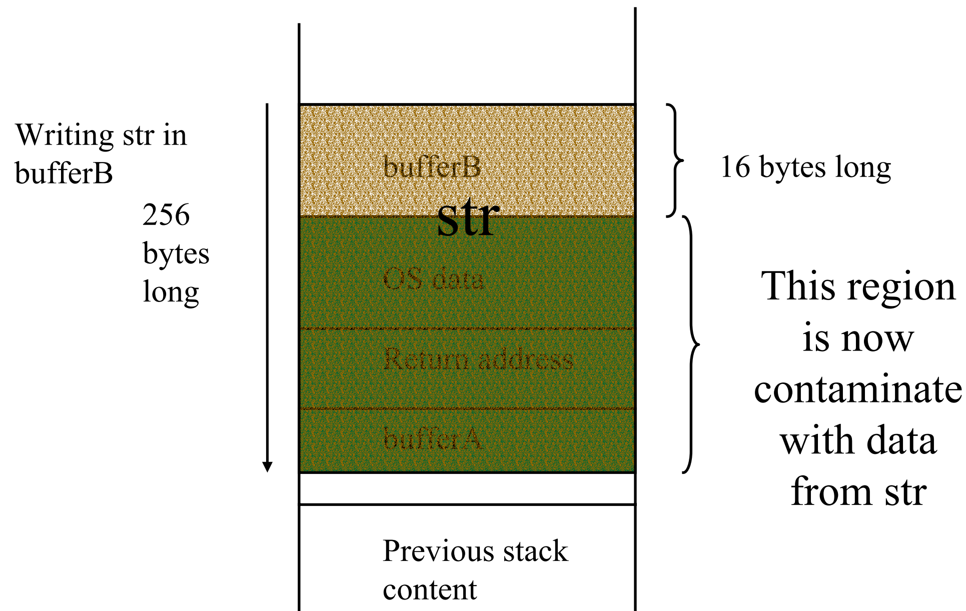
```
void myFunction(char *str) {
    char bufferB[16];
     strcpy(bufferB,str);

}
```



Buffer Overflowing Mechanism

As we execute strcpy() in myFunction(), the system begins to copy str into the space allocated for bufferB.

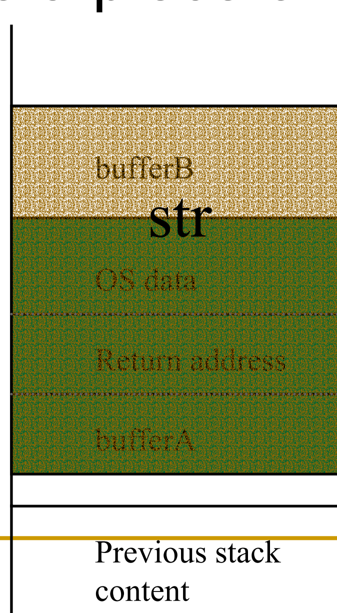
Since there is only 16 bytes of space for bufferB, the remaining data is written over the rest of the stack.



Buffer Overflowing Mechanism

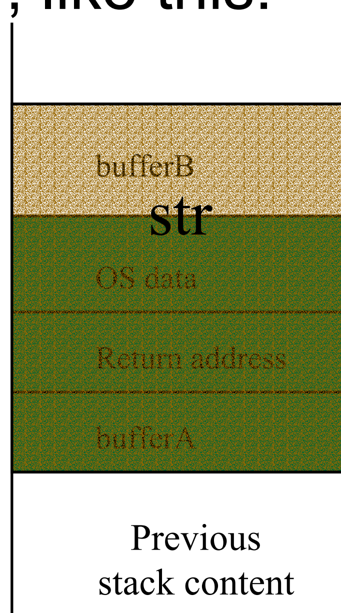
- As we can see, the stack as been contaminated with improper data.
- Some data has been written over the return address space in the stack.
- If the content of str is carefully selected, we can point the return address to a piece of code we have written.

If this piece of data is carefully selected, we can point the OS to go wherever we want.



Buffer Overflowing Mechanism

- Usually, in a buffer overflow attack, the return address is selected to point to the buffer that caused the overflow, like this:



When the system returns from the function call, it will begin executing code from the beginning of str

How Is Buffer Overflow Bad?

- Usually, the hacker wants to write code to gain access to the computer or gain more privileges on the system. Once this occurs, a number of system violations or damage can easily be performed
- Corrupts or damages program running, causing it to fail; produces incorrect results with other programs
- Can corrupt programs, causing it to disclose confidential information
- Can corrupt program, take remote control, and have it do undesired things

Example Buffer Overflow Code

```
Void CopyString(char *dest, char *source){
   while(*source){
               *dest++ = *source++;
   }
}
void Example (){
   char                buffer[16];
   CopyString(buffer, "This string is too long!");
}
```

A Possible Solution

```
Void CopyString(char *dest, char *source, int length){
    int i=0;
    while((*source) && (i < length)){
                *dest++ = *source++;
                i++;
    }
}
void Example (){
    char                buffer[16];
    CopyString(buffer, "This string is too long!", 16);
}
```