HOST VULNERABILITIES

UT CS361S – Network Security and Privacy

Fall 2021

Lecture Notes

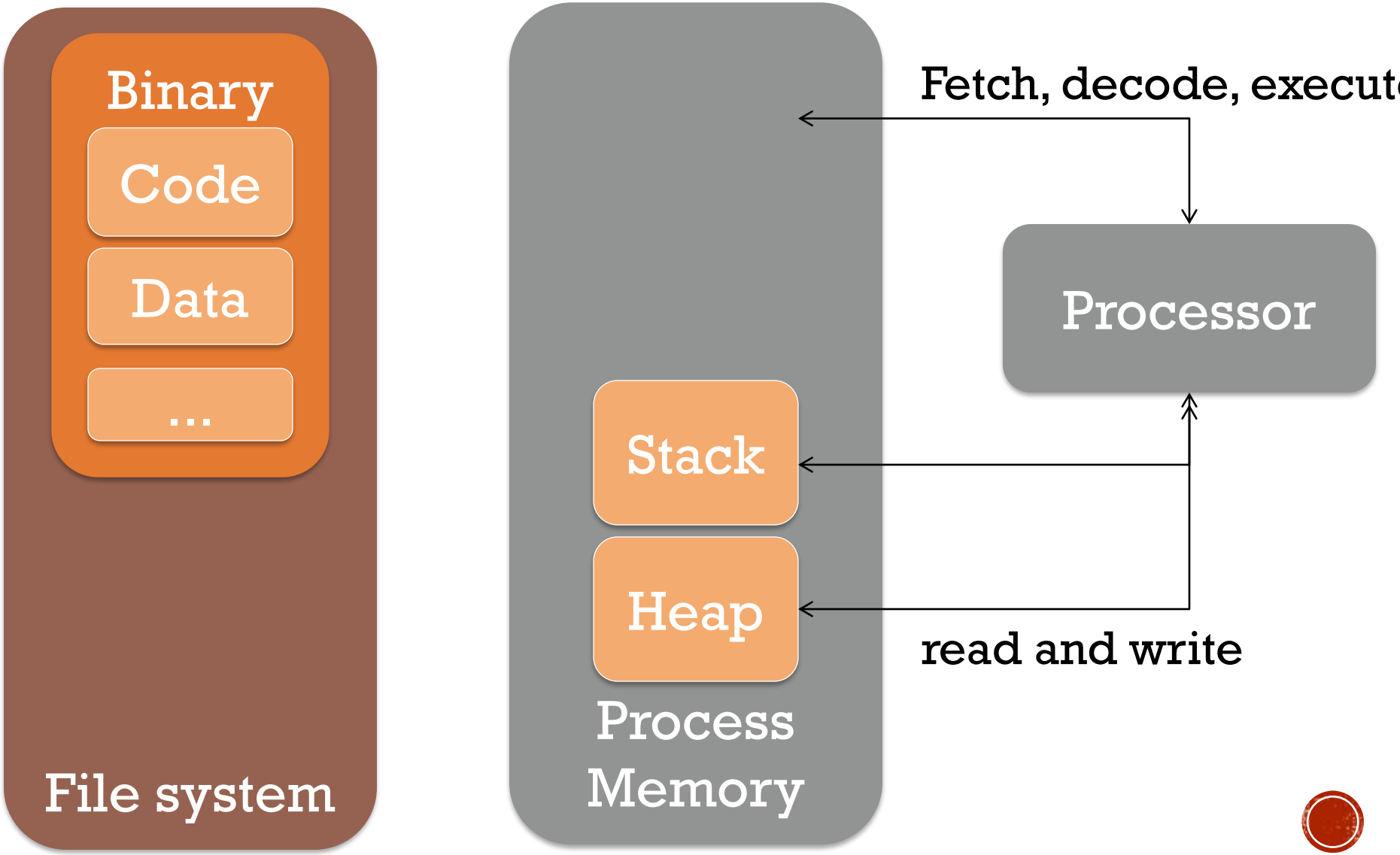
VULNERABILITIES AND NETSEC

- This class is "Network Security"
- What do host vulnerabilities have to do with it?
- Hosts are "nodes" in a network graph
- Vulnerabilities can be exploited by remote attackers
  - Either to directly access resources on a particular host
  - Or, to penetrate network defenses and access a more valuable host

BRIEF OVERVIEW TO EXECUTION

- ***Very brief*** overview of ***Control Flow Hijacking***
  - There are other types of vulnerabilities (e.g misconfigured)
  - Control Flow Hijacking is probably the hardest to grasp
- Critical Concepts:
  - The "normal" flow of control for authorized instructions
  - Inputs that change the flow to unauthorized instructions
- ATTRIBUTION: Derived from slides by Dave Brumley, CMU

BASIC EXECUTION



SETH'S NOTES

- Stack
  - For temporary static variables
  - Function call/return data
  - Linear
  - Generally, tightly managed
- Heap
  - Global variables and dynamic variables
  - Hierarchical, "free floating"
  - Fragmented, not tightly managed

SETH'S NOTES

- Assembly "function calls" don't really exist
  - Rather, jump to new location ("function")
  - Save context of old location
  - Load context for new location
  - Include information for "returning"

SETH'S NOTES

- There are multiple ways to do this
- "Calling Conventions"
- Caller Cleanup – caller cleans stack
- Callee Cleanup – called function cleans stack
- Other convention variations:
  - Order that function data is loaded onto stack
  - Whether some data is put into registers instead

SETH'S NOTES

- Visualizing caller v callee cleanup

```
    push arg1
    push arg2
    push arg3
    call proc

proc:
    pop  r1   ; the return address
    pop  r2
    pop  r2
    pop  r2
    push r1
    ret
```

:decl (ca

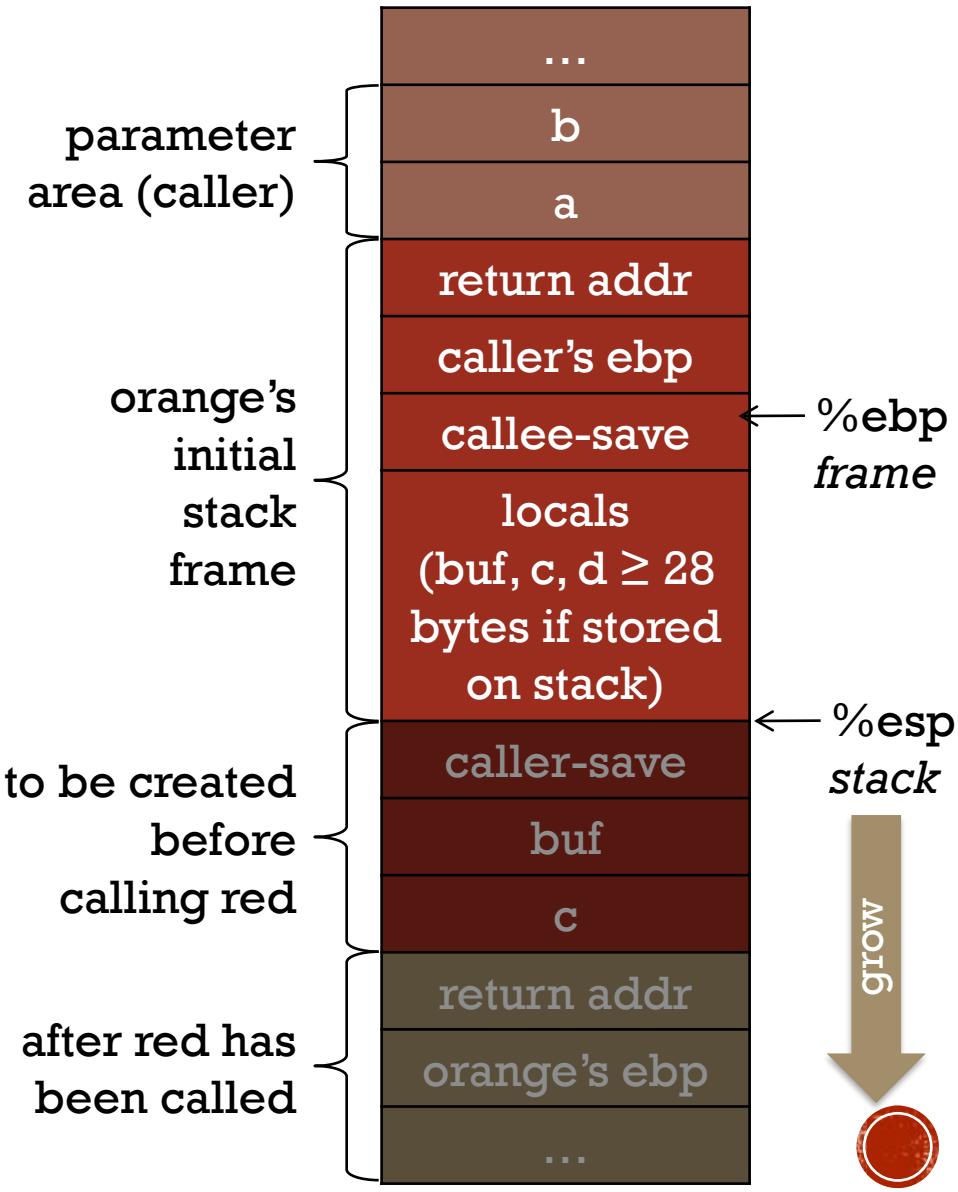
```
    push arg1
    push arg2
    push arg3
    call proc
    pop  r2
    pop  r2
    pop  r2

proc:
    ret
```

EBP AND ESP

- EBP
  - Stack Base Pointer
  - Where the stack was when the routine started
- ESP
  - Stack Pointer
  - Top of the current stack
- EBP is a previous function's saved ESP

CDECL – DEFAULT FOR LINUX & GCC

```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



GDB WALKTHROUGH

- SRC: tenouk.com/Bufferoverflowc/Bufferoverflow3.html
- Given C code, examine assembly via GDB
- Uses cdecl calling convention

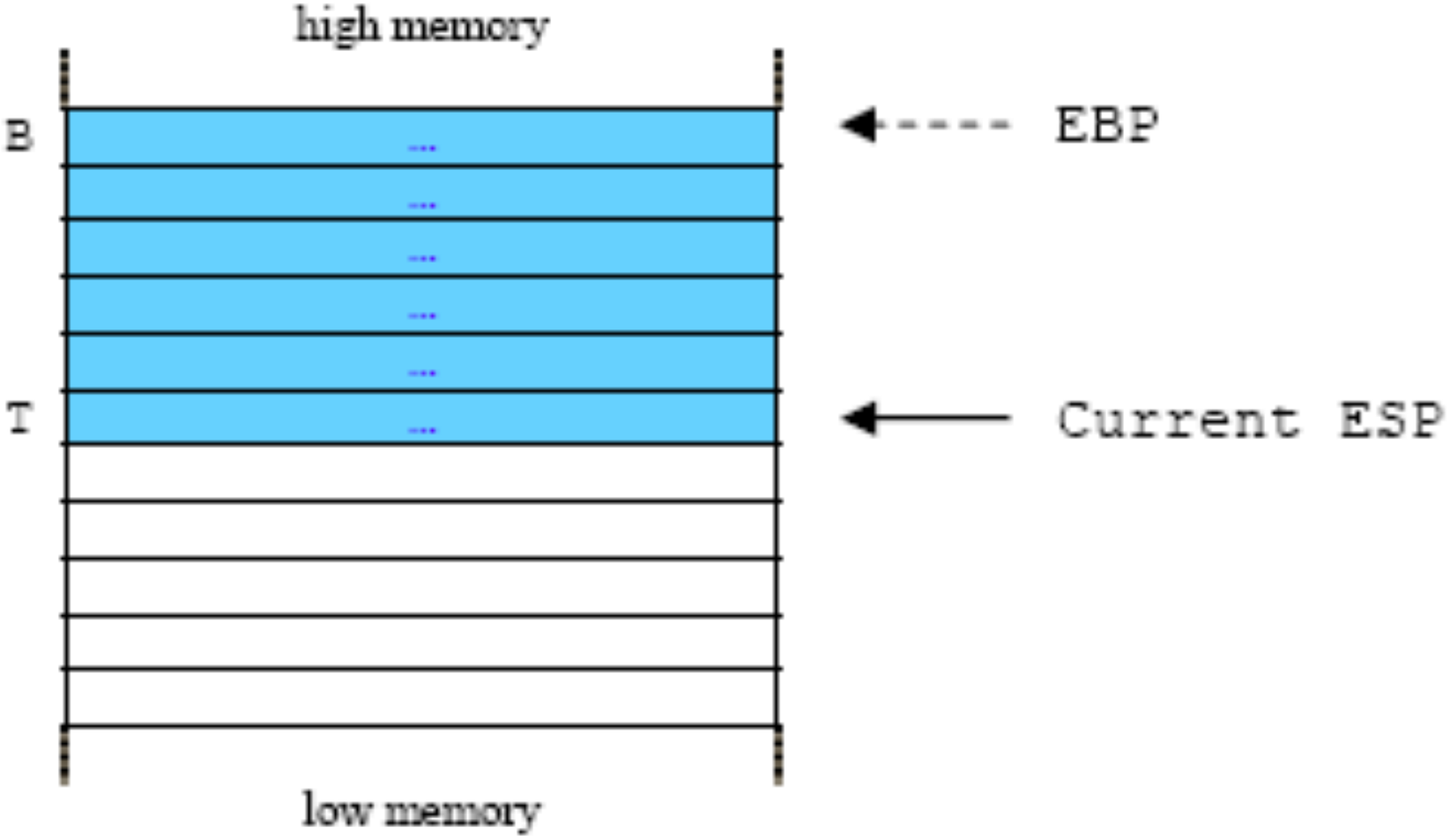
GDB WALKTHROUGH – C CODE

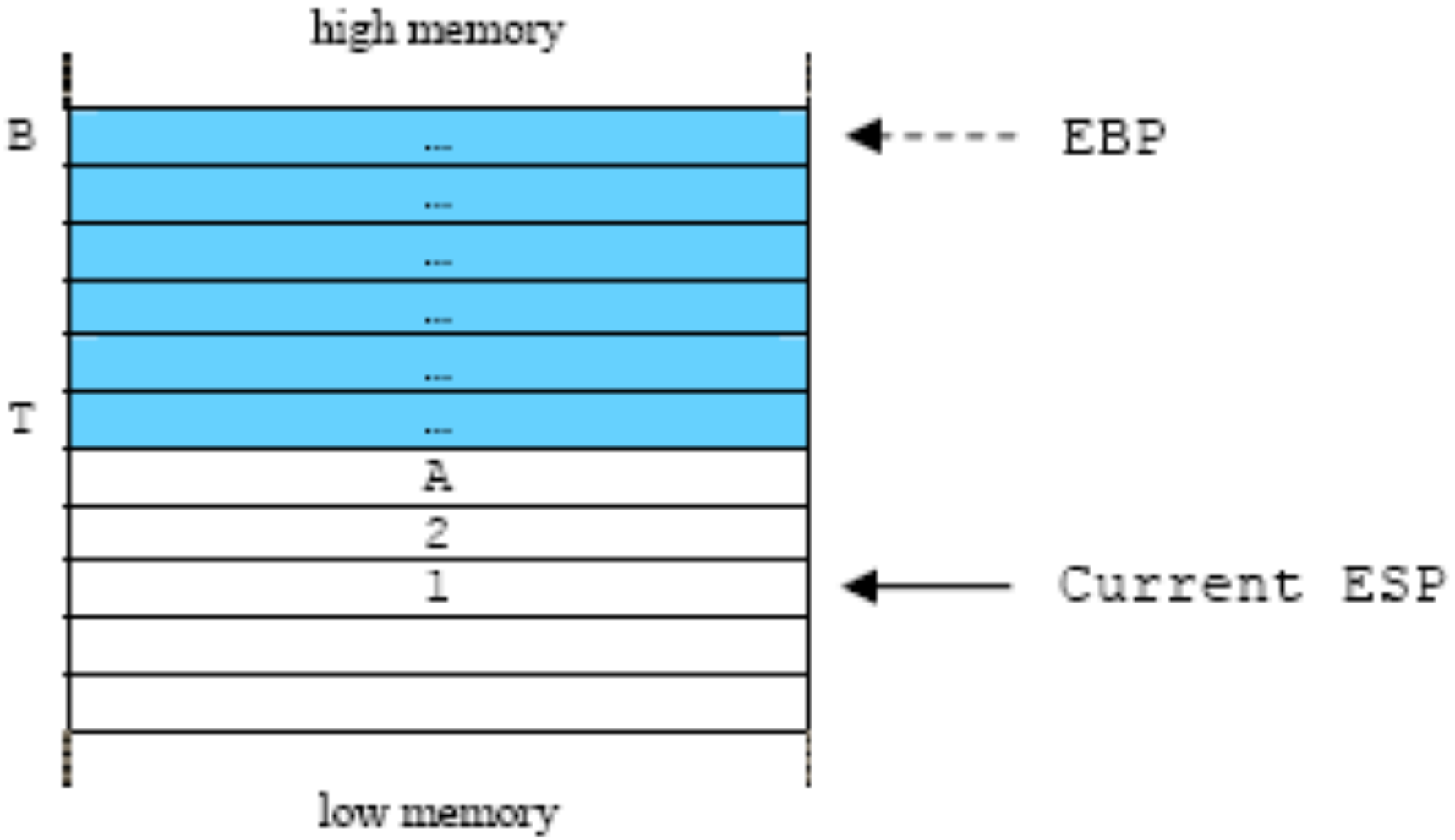
```
#include <stdio.h>

int TestFunc(int parameter1,int parameter2,char parameter3)
{
int y = 3, z = 4;
char buff[7] = "ABCDEF";

// function's task code here
return 0;
}

int main(int argc, char *argv[ ])
{
TestFunc(1, 2, 'A');
return 0;
}
```

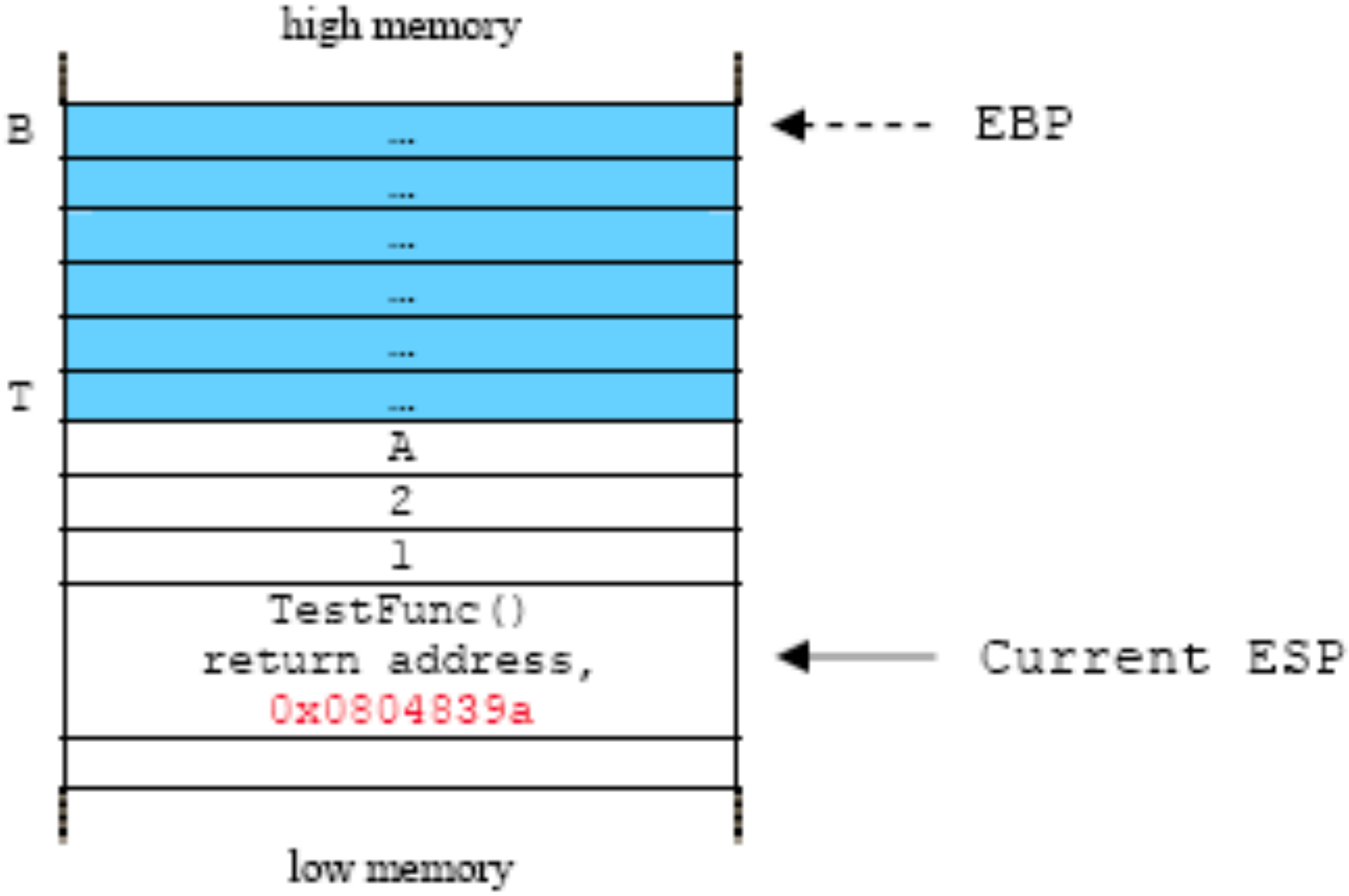
GDB WALKTHROUGH – CALL TESTFUNC

Register eax loaded with character 'A' (0x41), not shown

```
0x08048390 <main+36>:   push   %eax      ;push the third parameter, 'A' prepared in eax onto the stack, [ebp+16]
0x08048391 <main+37>:   push   $0x2      ;push the second parameter, 2 onto the stack, [ebp+12]
0x08048393 <main+39>:   push   $0x1      ;push the first parameter, 1 onto the stack, [ebp+8]
-------------------------------------------------------------------------------------------------------------------------
0x08048395 <main+41>:   call   0x8048334 <TestFunc>   ;function call. Push the return
                                                      ;address [0x0804839a] onto the stack, [ebp+4]
```


high memory
B
T
EBP
Current ESP
low memory


high memory
B
T
...
...
...
...
...
...
A
2
1
EBP
Current ESP
low memory


high memory
B
T
...
...
...
...
...
...
A
2
1
TestFunc()
return address,
0x0804839a
EBP
Current ESP
low memory

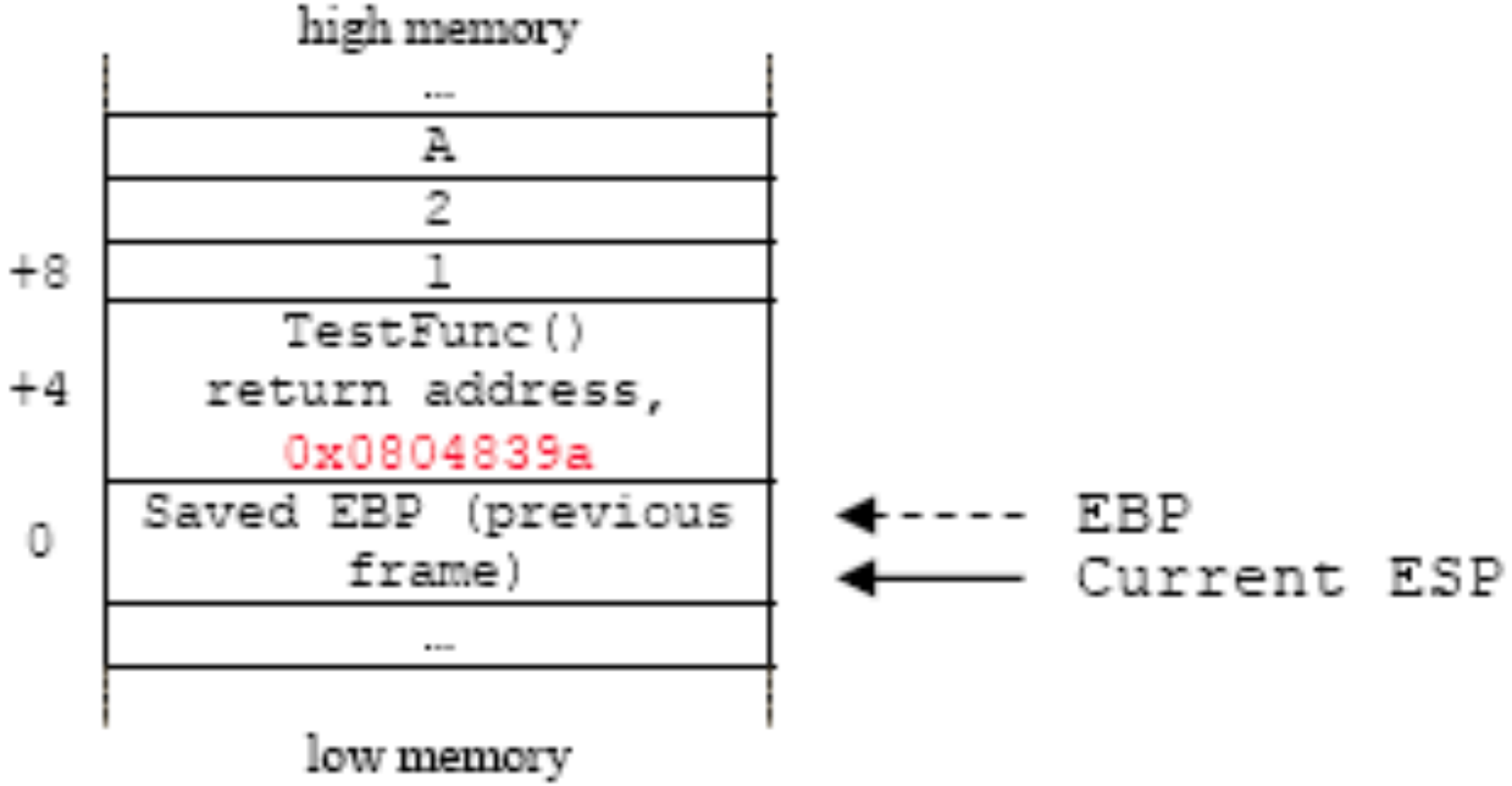
GDB WALKTHROUGH – TESTFUNC() C CODE

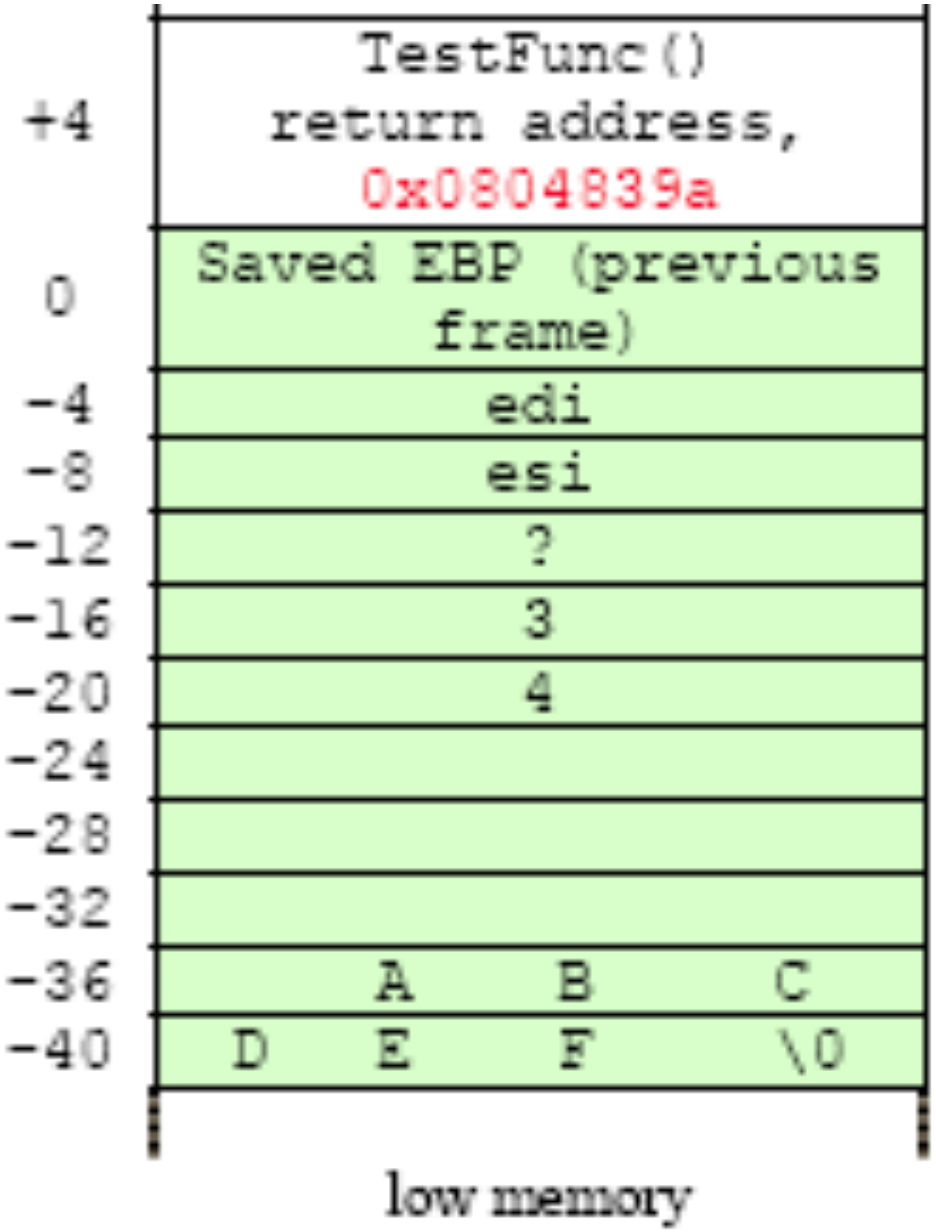
```
int TestFunc(int parameter1,int parameter2,char parameter3)
{
int y = 3, z = 4;
char buff[7] = "ABCDEF";

// function's task code here
return 0;
}
```

GDB WALKTHROUGH – TESTFUNC() ASSEMBLY

```
0x08048334 <TestFunc+0>:        push   %ebp          ;push the previous stack frame
                                                      ;pointer onto the stack, [ebp+0]
0x08048335 <TestFunc+1>:        mov    %esp, %ebp    ;copy the ebp into esp, now the ebp and esp
                                                      ;are pointing at the same address,
                                                      ;creating new stack frame [ebp+0]
0x08048337 <TestFunc+3>:        push   %edi          ;save/push edi register, [ebp-4]
0x08048338 <TestFunc+4>:        push   %esi          ;save/push esi register, [ebp-8]
0x08048339 <TestFunc+5>:        sub    $0x20, %esp   ;subtract esp by 32 bytes for local
                                                      ;variable and buffer if any, go to [ebp-40]
```

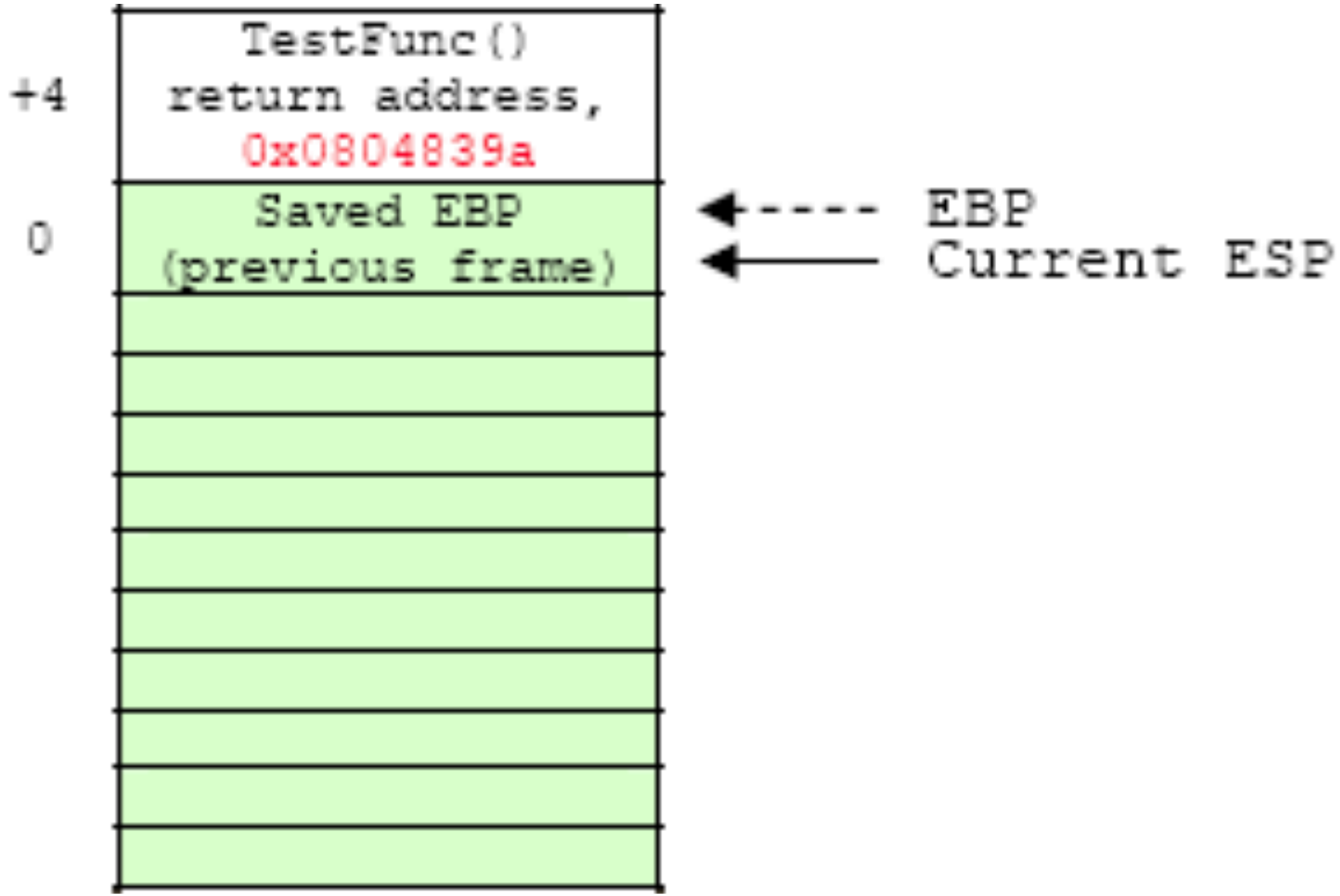
32 bytes allocated on stack (0x20). Variables Loaded into this space (not shown).


high memory
...
A
2
+8
1
TestFunc()
+4
return address,
0x0804839a
Saved EBP (previous
0
frame)
EBP
Current ESP
...
low memory


TestFunc()
+4
return address,
0x0804839a
Saved EBP (previous
0
frame)
EBP
-4
edi
Save Registers
-8
esi
-12
?
-16
3
Local variables 'y' and 'z'
-20
4
-24
-28
-32
-36
A
B
C
buff[7] (note, more allocate
-40
D
E
F
\0
Current ESP
low memory

GDB WALKTHROUGH – TESTFUNC() EXIT

```
0x08048365 <TestFunc+49>:    add    $0x20, %esp     ;add 32 bytes to esp, back to [ebp-8]
0x08048368 <TestFunc+52>:    pop    %esi           ;restore the esi, [ebp-4]
0x08048369 <TestFunc+53>:    pop    %edi           ;restore the edi, [ebp+0]
```

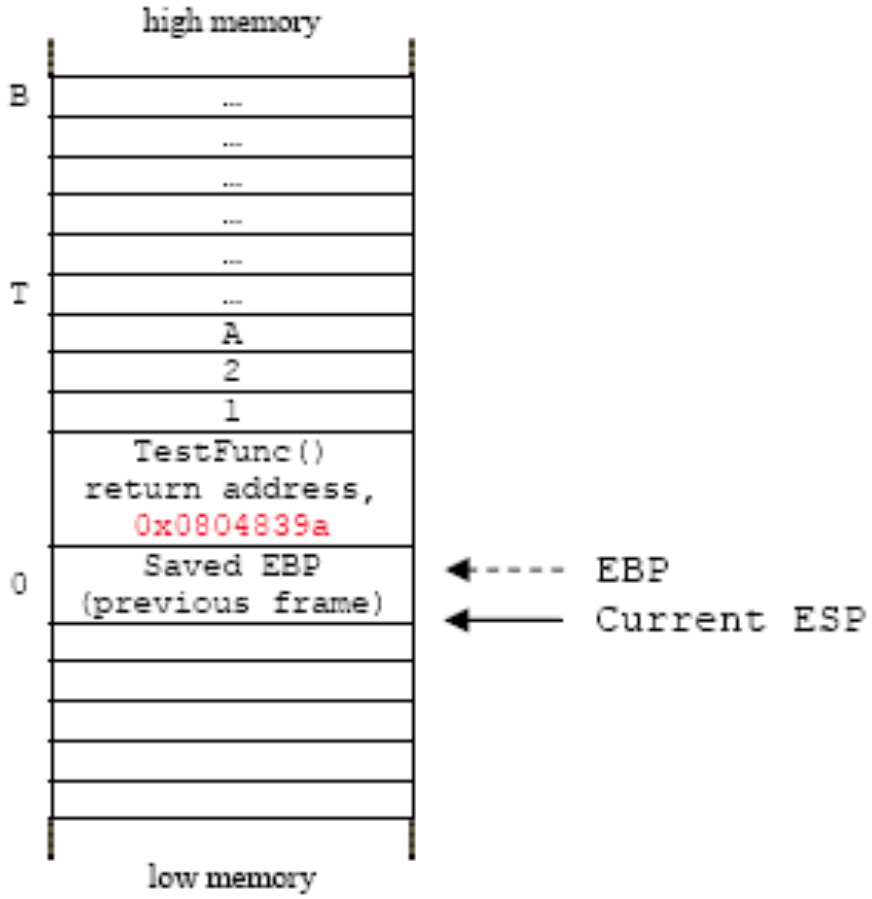

TestFunc()
+4
return address,
0x0804839a
Saved EBP
0
(previous frame)
EBP
Current ESP

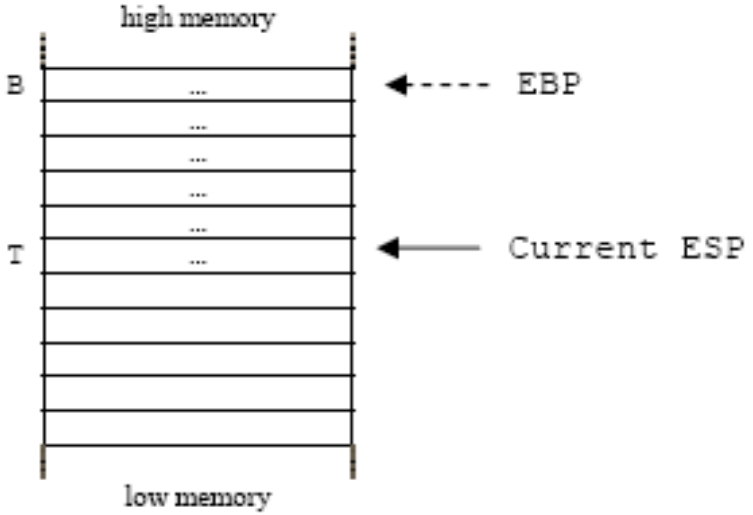
GDB WALKTHROUGH – TESTFUNC() EXIT, PART 2

```
0x0804836a <TestFunc+54>:    leave   ;restoring the ebp to the previous stack frame, [ebp+4]
0x0804836b <TestFunc+55>:    ret     ;transfer control back to calling function using
                                      ;the saved return address at [ebp+8]
```

GDB WALTHROUGH – MAIN() AFTER TESTFUNC() RETURN

```
0x0804839a <main+46>:    add    $0xc, %esp    ;cleanup the 3 parameters pushed on the stack
                                                ;at [ebp+8], [ebp+12] and [ebp+16]
                                               ;total up is 12 bytes = 0xc
```


high memory
B
...
...
...
...
...
T
...
A
2
1
TestFunc()
return address,
0x0804839a
0
Saved EBP
(previous frame)
EBP
Current ESP
low memory


high memory
B
...
...
...
...
...
T
...
EBP
Current ESP
low memory

WHAT ARE BUFFER OVERFLOWS?

A ***buffer overflow*** occurs when data is written outside of the space allocated for the buffer.

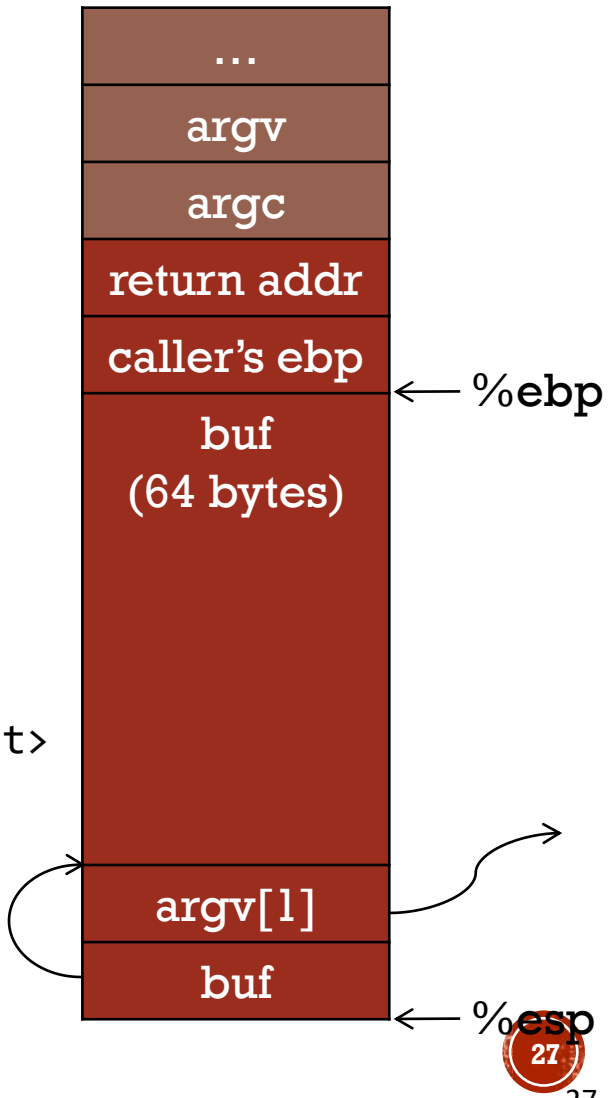
- C does not check that writes are in-bound

1. Stack-based
   - covered in this class
2. Heap-based
   - more advanced
   - very dependent on system and library version

BASIC EXAMPLE

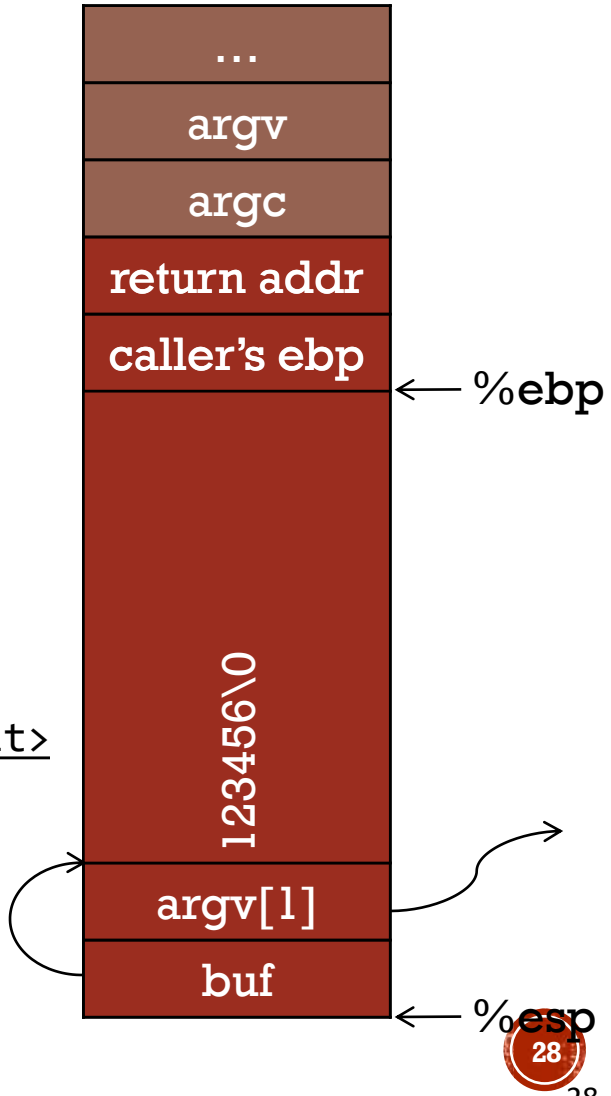
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $72,%esp
   0x080483ea <+6>:     mov    12(%ebp),%eax
   0x080483ed <+9>:     mov    4(%eax),%eax
   0x080483f0 <+12>:    mov    %eax,4(%esp)
   0x080483f4 <+16>:    lea    -64(%ebp),%eax
   0x080483f7 <+19>:    mov    %eax,(%esp)
   0x080483fa <+22>:    call   0x8048300 <strcpy@plt>
   0x080483ff <+27>:    leave
   0x08048400 <+28>:    ret
```



"123456"

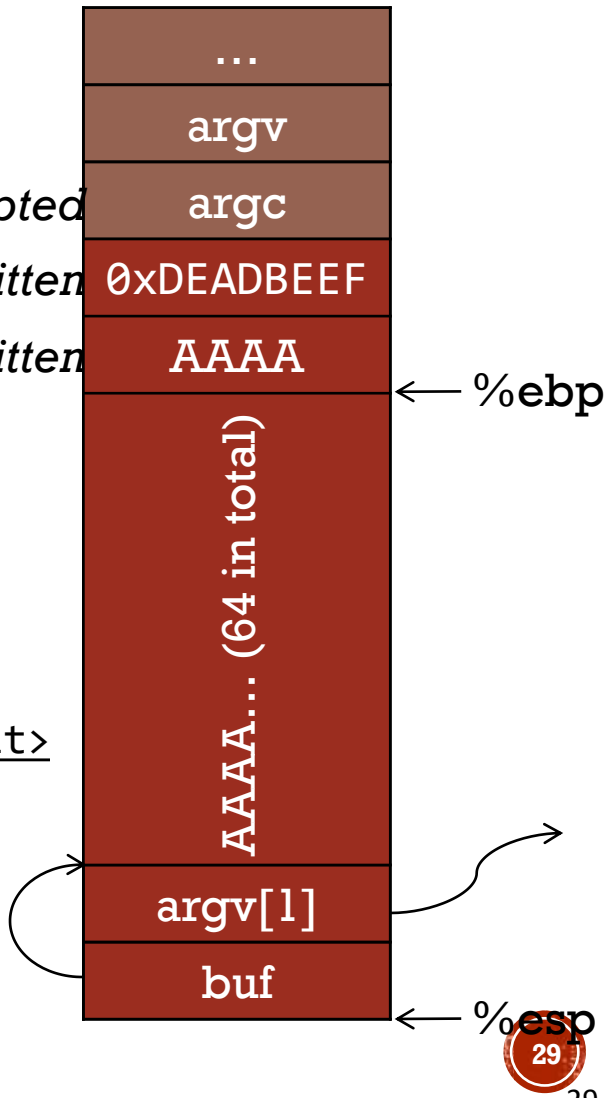
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $72,%esp
   0x080483ea <+6>:     mov    12(%ebp),%eax
   0x080483ed <+9>:     mov    4(%eax),%eax
   0x080483f0 <+12>:    mov    %eax,4(%esp)
   0x080483f4 <+16>:    lea    -64(%ebp),%eax
   0x080483f7 <+19>:    mov    %eax,(%esp)
   0x080483fa <+22>:    call   0x8048300 <strcpy@plt>
   0x080483ff <+27>:    leave
   0x08048400 <+28>:    ret
```



"A"X68 ."\XEF\XBE\XAD\XDE"

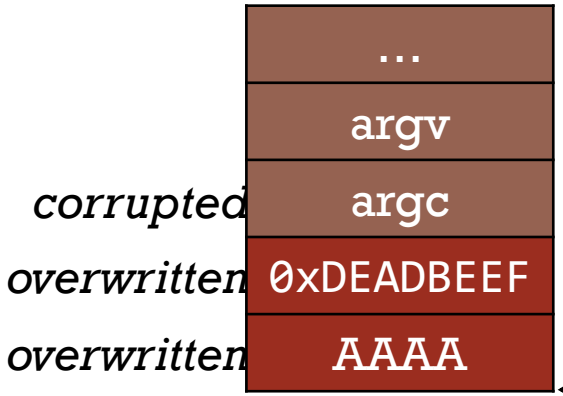
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $72,%esp
   0x080483ea <+6>:     mov    12(%ebp),%eax
   0x080483ed <+9>:     mov    4(%eax),%eax
   0x080483f0 <+12>:    mov    %eax,4(%esp)
   0x080483f4 <+16>:    lea    -64(%ebp),%eax
   0x080483f7 <+19>:    mov    %eax,(%esp)
   0x080483fa <+22>:    call   0x8048300 <strcpy@plt>
   0x080483ff <+27>:    leave
   0x08048400 <+28>:    ret
```



FRAME TEARDOWN—1

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $72,%esp
   0x080483ea <+6>:     mov    12(%ebp),%eax
   0x080483ed <+9>:     mov    4(%eax),%eax
   0x080483f0 <+12>:    mov    %eax,4(%esp)
   0x080483f4 <+16>:    lea    -64(%ebp),%eax
   0x080483f7 <+19>:    mov    %eax,(%esp)
   0x080483fa <+22>:    call   0x8048300 <strcpy(
=> 0x080483ff <+27>:    leave
   0x08048400 <+28>:    ret
```



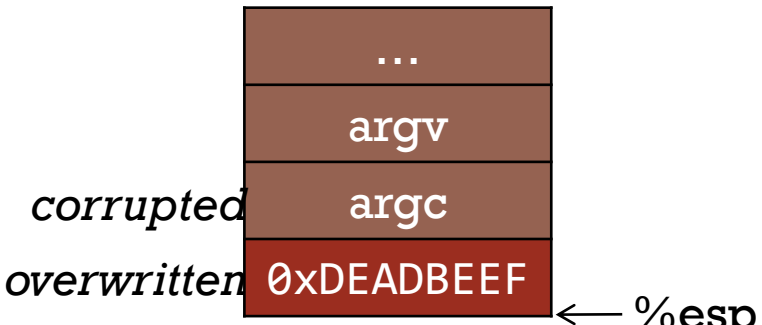

```
leave
1. mov %ebp,%esp
2. pop %ebp
```

← %esp

FRAME TEARDOWN—2

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $72,%esp
   0x080483ea <+6>:     mov    12(%ebp),%eax
   0x080483ed <+9>:     mov    4(%eax),%eax
   0x080483f0 <+12>:    mov    %eax,4(%esp)
   0x080483f4 <+16>:    lea    -64(%ebp),%eax
   0x080483f7 <+19>:    mov    %eax,(%esp)
   0x080483fa <+22>:    call   0x8048300 <strcpy(
   0x080483ff <+27>:    leave
   0x08048400 <+28>:    ret
```



%ebp = AAAA

leave
1. mov %ebp,%esp
2. pop %ebp

31

FRAME TEARDOWN—3

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```


```
Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $72,%esp
   0x080483ea <+6>:     mov    12(%ebp),%eax
   0x080483ed <+9>:     mov    4(%eax),%eax
   0x080483f0 <+12>:    mov    %eax,4(%esp)
   0x080483f4 <+16>:    lea    -64(%ebp),%eax
   0x080483f7 <+19>:    mov    %eax,(%esp)
   0x080483fa <+22>:    call   0x8048300 <strcpy@plt>
   0x080483ff <+27>:    leave
   0x08048400 <+28>:    ret
```

%eip = 0xDEADBEEF
(probably crash)

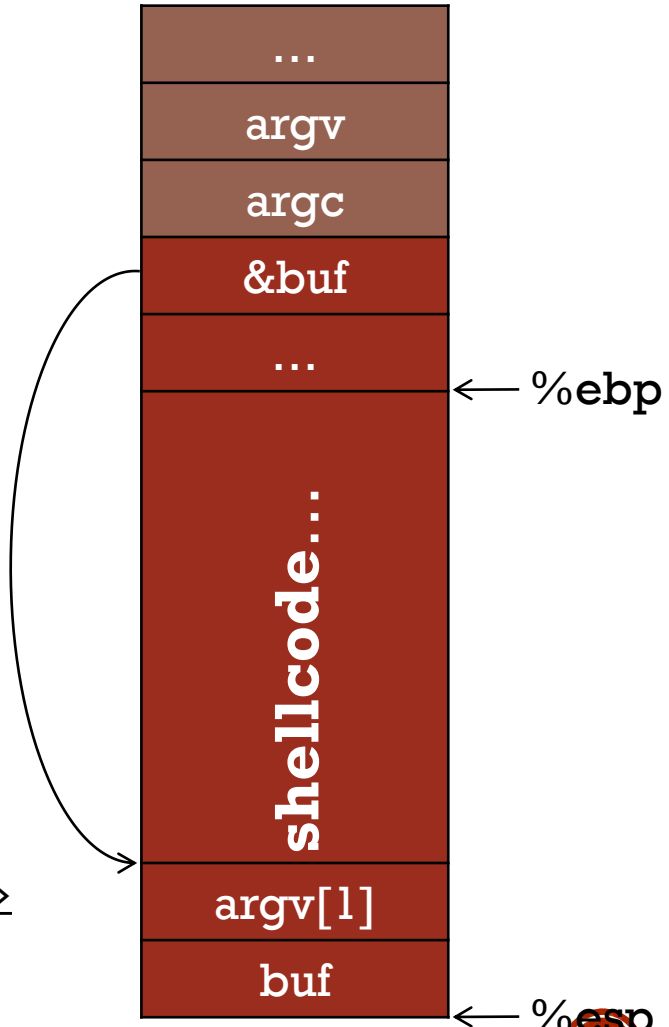
32

SHELLCODE

Traditionally, we inject assembly instructions for `exec("/bin/sh")` into buffer.

- see "*Smashing the stack for fun and profit*" for exact string
- or search online

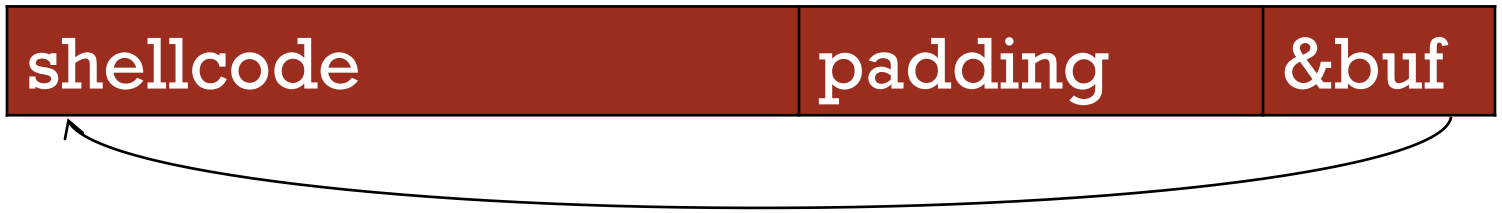
```
…
0x080483fa <+22>: call   0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```



RECAP

To generate ***exploit*** for a basic buffer overflow:

1. Determine size of stack frame up to head of buffer
2. Overflow buffer with the right size



computation + ***control***

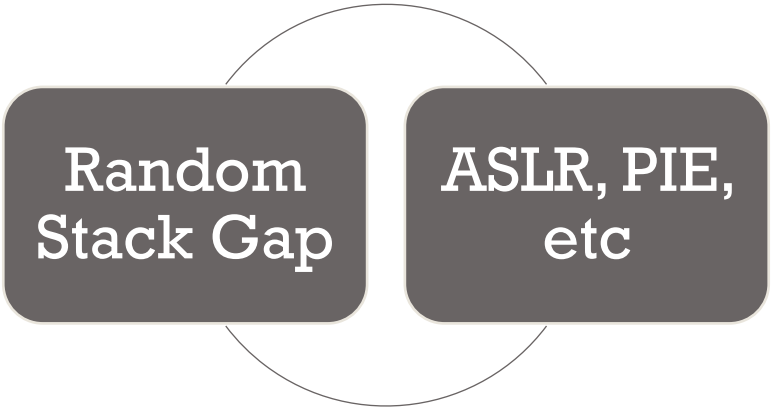
DEALING WITH CONTROL FLOW VIOLATIONS

Make it harder to control a subverted flow

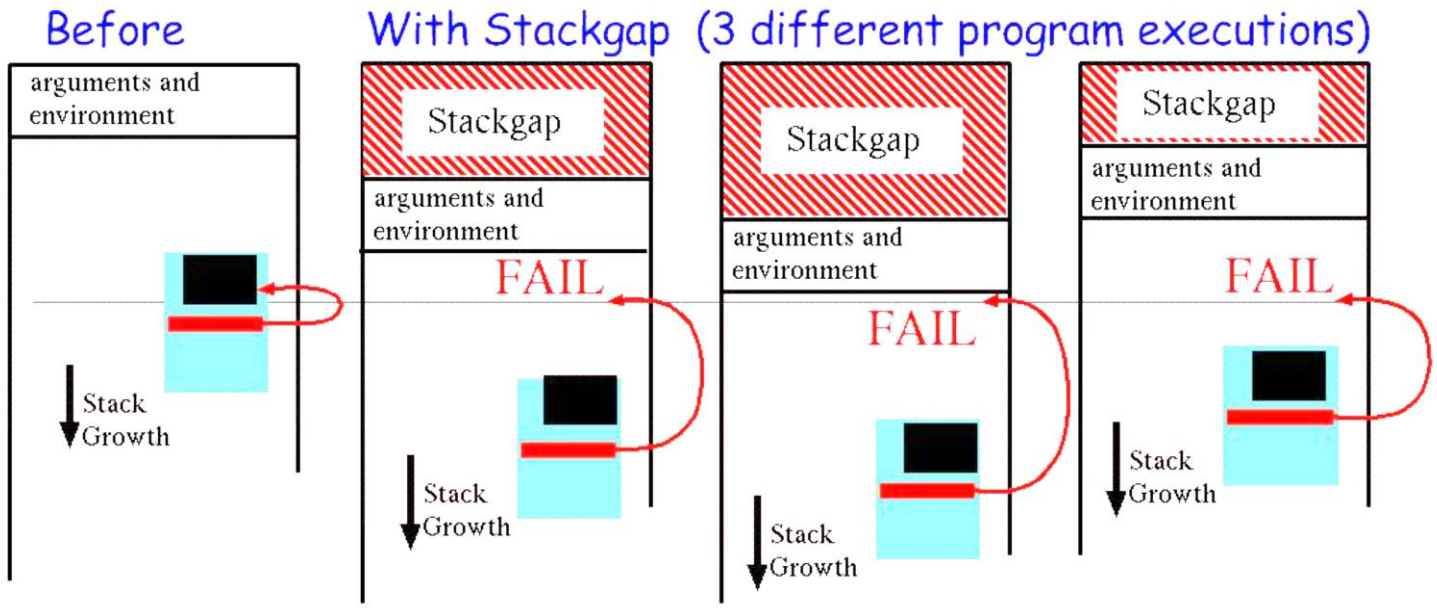
Make taking control of the flow innocuous

Make it harder to get control of the flow

DISRUPTING EXPLOITATIVE OPERATIONS



RANDOM STACK GAP

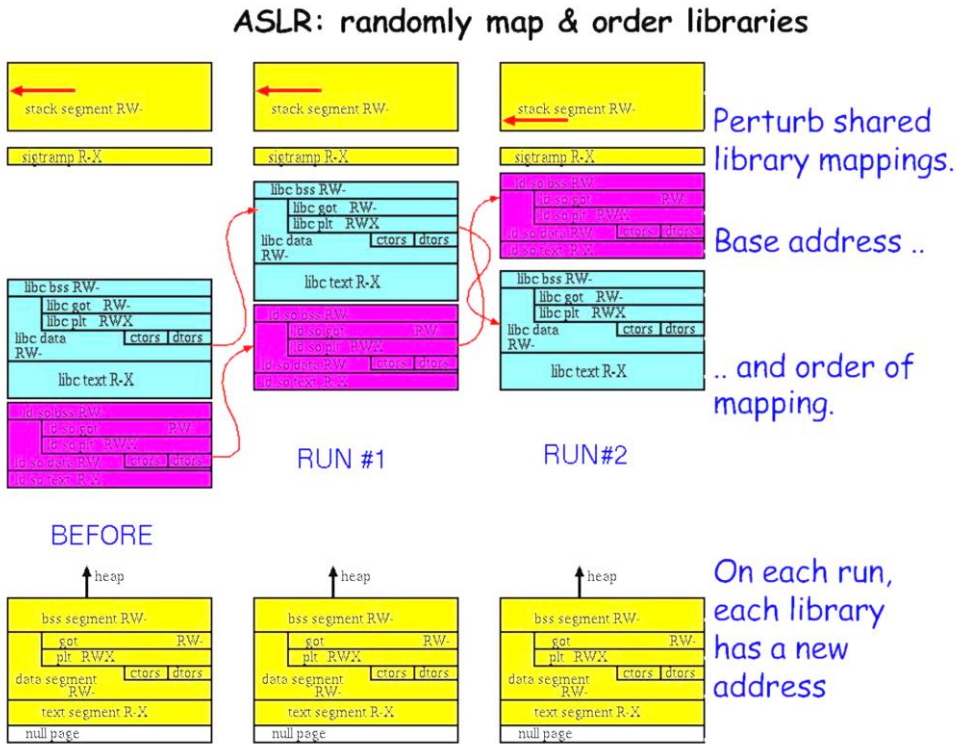


ASLR

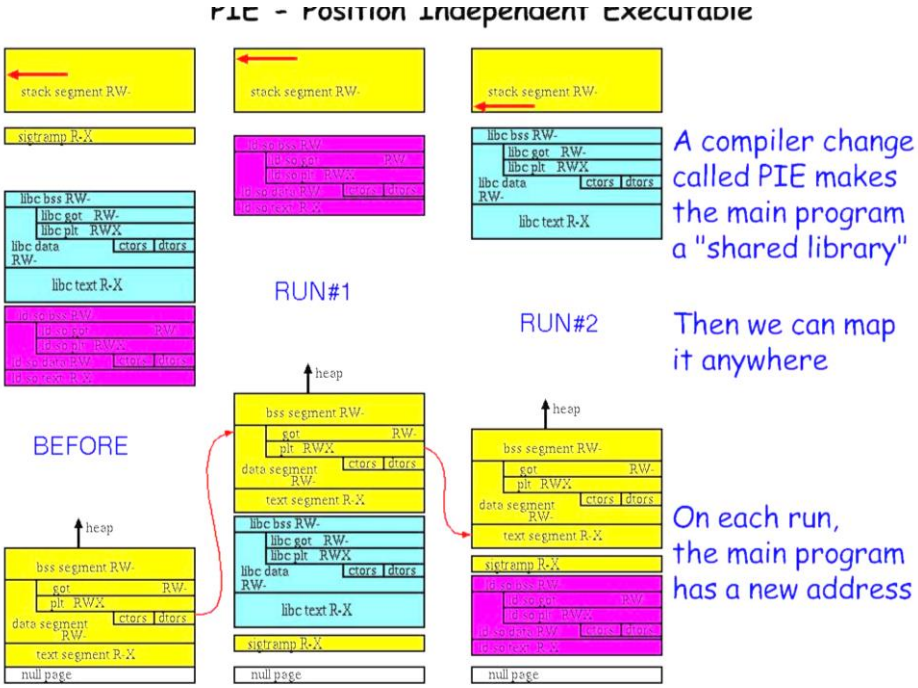
Address Space Layout Randomization

- Subversion usually needs to know memory layout
- General goal: make layout unpredictable

START WITH LIBRARIES



ADD EXECUTABLES



FINALLY, DYNAMIC ALLOCATIONS

mmap

malloc

LIMITATIONS OF ASLR

1. **Boot-time based randomization**
2. **Unsupported executables/libraries, low-entropy.**
3. **ASLR does not *trap* the attack**
4. **ASLR does not alert in a case of an attack**
5. **ASLR does not *provide information* about an attack**
6. **ASLR is being bypassed by exploits daily**

Posted by **MORDECHAI GURI, PH.D.** on December 17, 2015

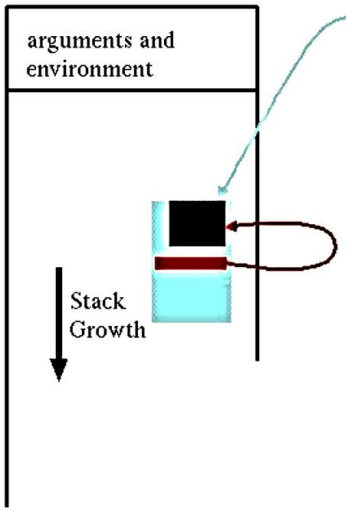
MAKING VIOLATIONS LESS DANGEROUS

W^X Permissions

rodata

W | X PERMISSIONS

Many bugs are exploitable because the address space has memory that is both writeable and executable (permissions = W | X)



this location has to be executable for the exploit to work

We could make the stack non-executable...

Hmmmm... how about a generic policy for the whole address space:

A page may be either writeable or executable, but not both (unless the program specifically requests)

We call this policy W ^ X (W xor X)

THE .RODATA SEGMENT

W^X Transition: The .rodata segment

Readonly strings and pointers were stored in the .text segment: X | R

Meaning const data could be executed (could be code an attacker could use as ROP payload)

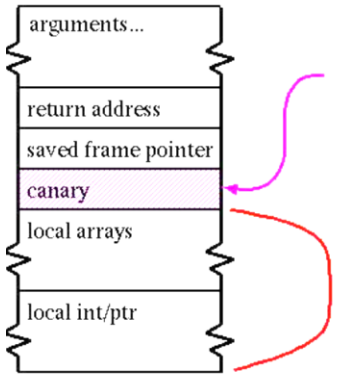
Solution: start using the ELF .rodata segment

These objects are now only R, lost their X permission

Greater policy: "minimal set of permissions"

FINALLY, BLOCKING EXPLOITS

Stack Protector



A typical stack frame...

Random value is inserted here by function prologue ...

... and checked by function epilogue

Reordering: Arrays (strings) placed closer to random value -- integers and pointers placed further away

-fstack-protector-all compiled system is 1.3% slower at make build

return-Oriented Programming

David Brumley
Carnegie Mellon University

Credit: Some slides from Ed Schwartz

ROP OVERVIEW

Idea:

We forge shell code out of existing application logic gadgets

Requirements:

vulnerability + gadgets + some *unrandomized or predictable* code (we need to know the addresses of gadgets)

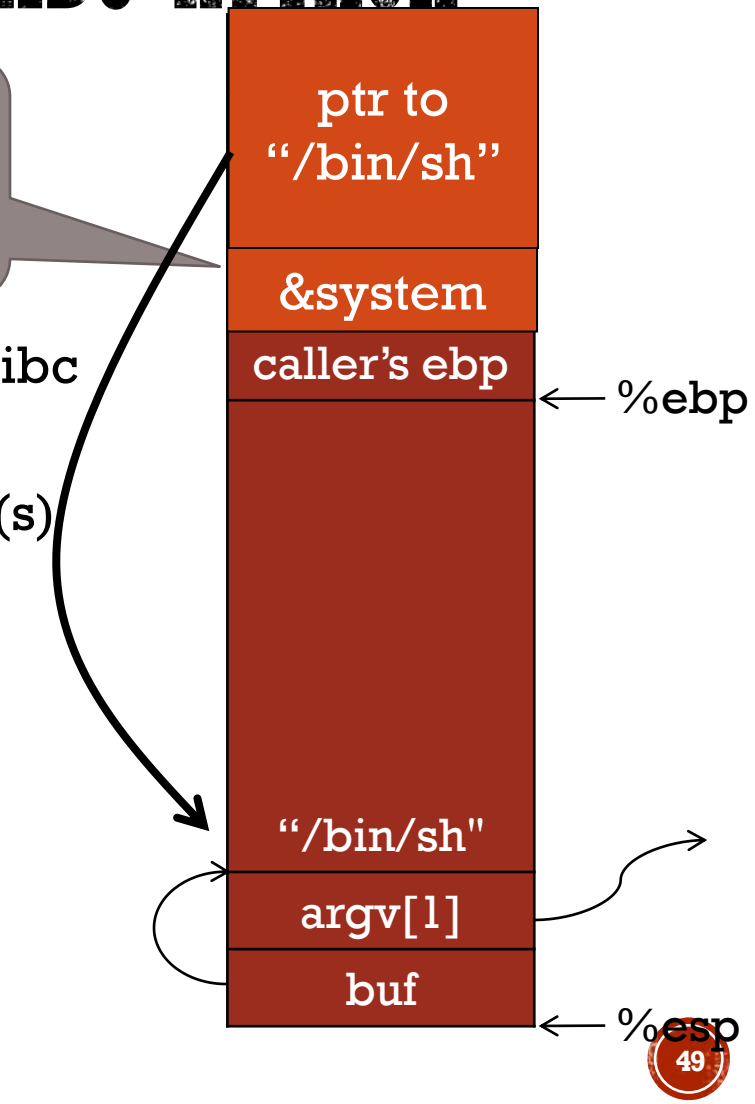
MOTIVATION: RETURN-TO-LIBC ATTACK

ret transfers control to `system`, which finds arguments on stack

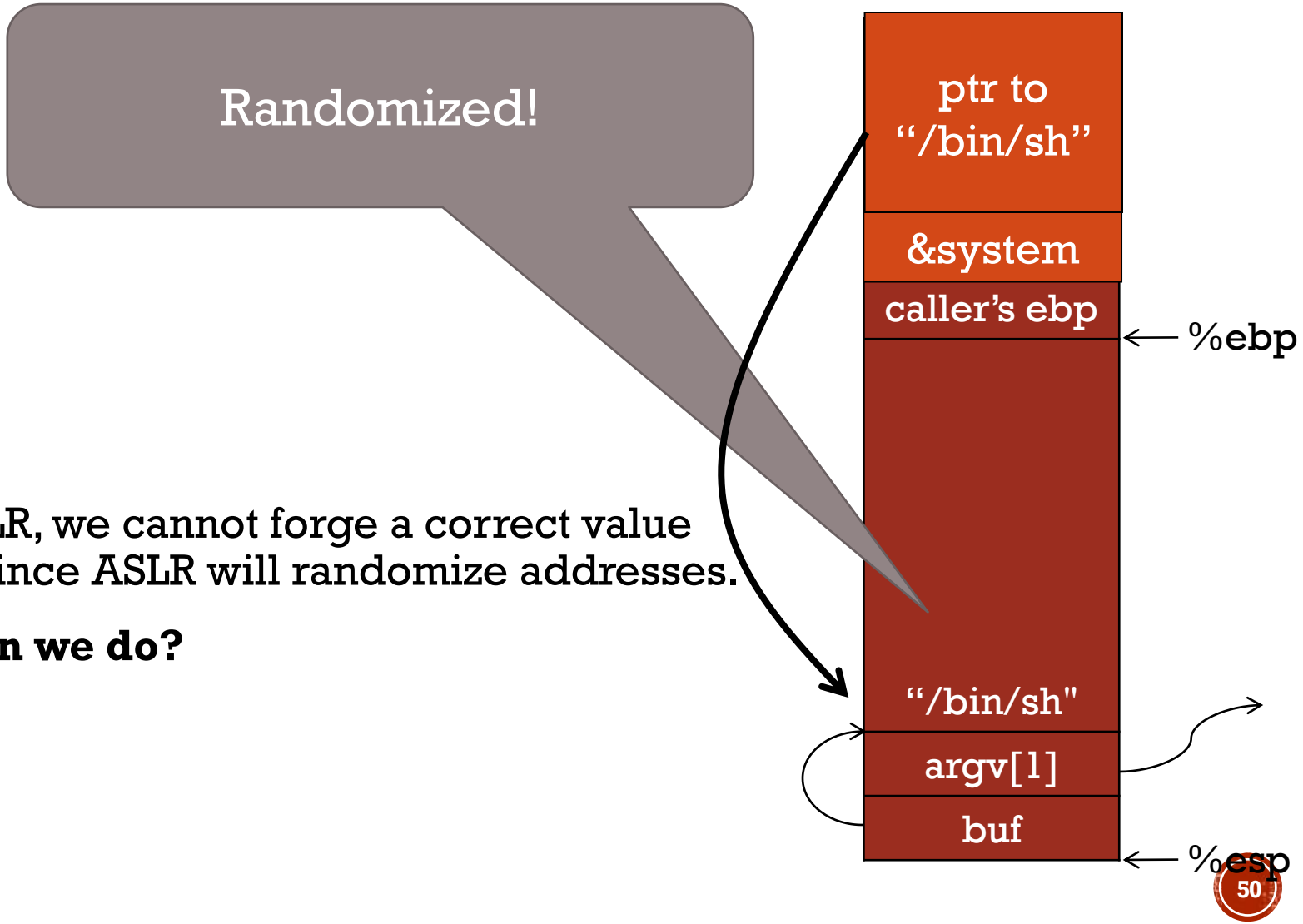
Overwrite return address with *address* of libc "system" function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!



QUESTION



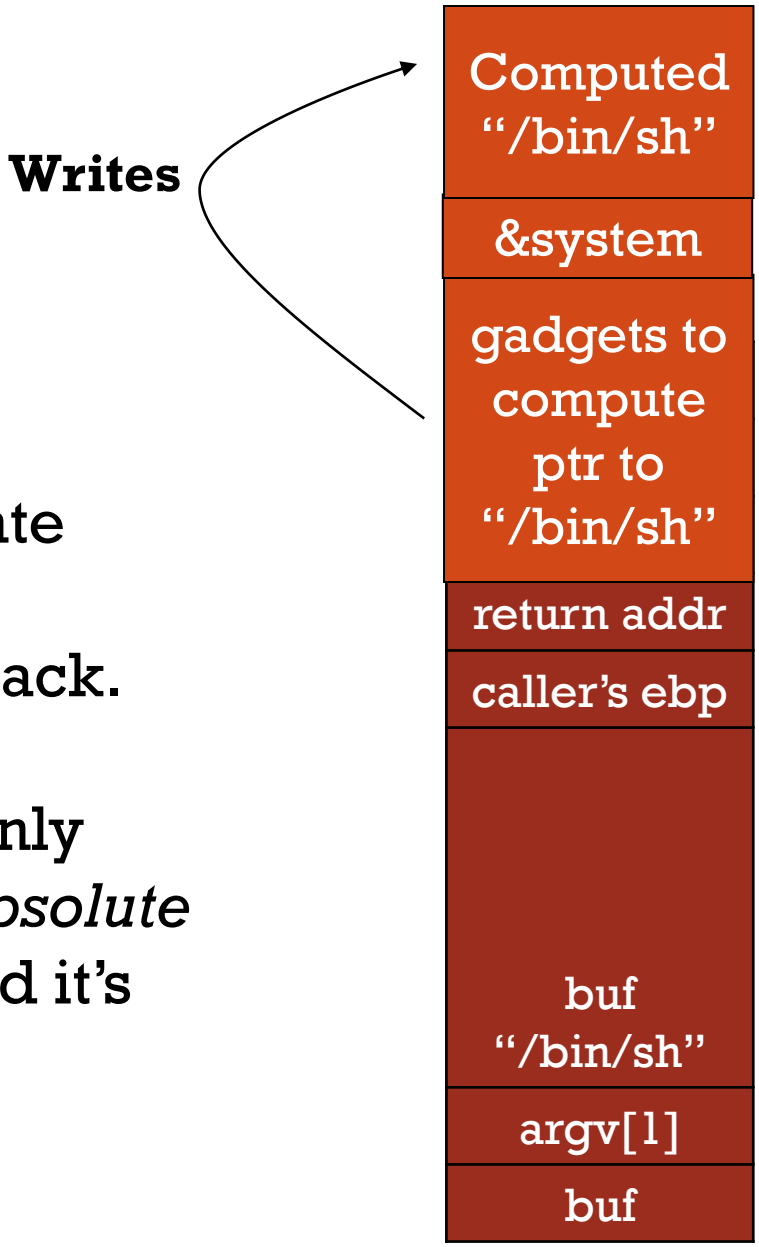
With ASLR, we cannot forge a correct value for `ptr` since ASLR will randomize addresses.

What can we do?

Idea!

Get a copy of ESP to calculate address of "/bin/sh" on randomized stack.

This works because ASLR only protects against knowing *absolute* addresses, while we will find it's *relative address*.

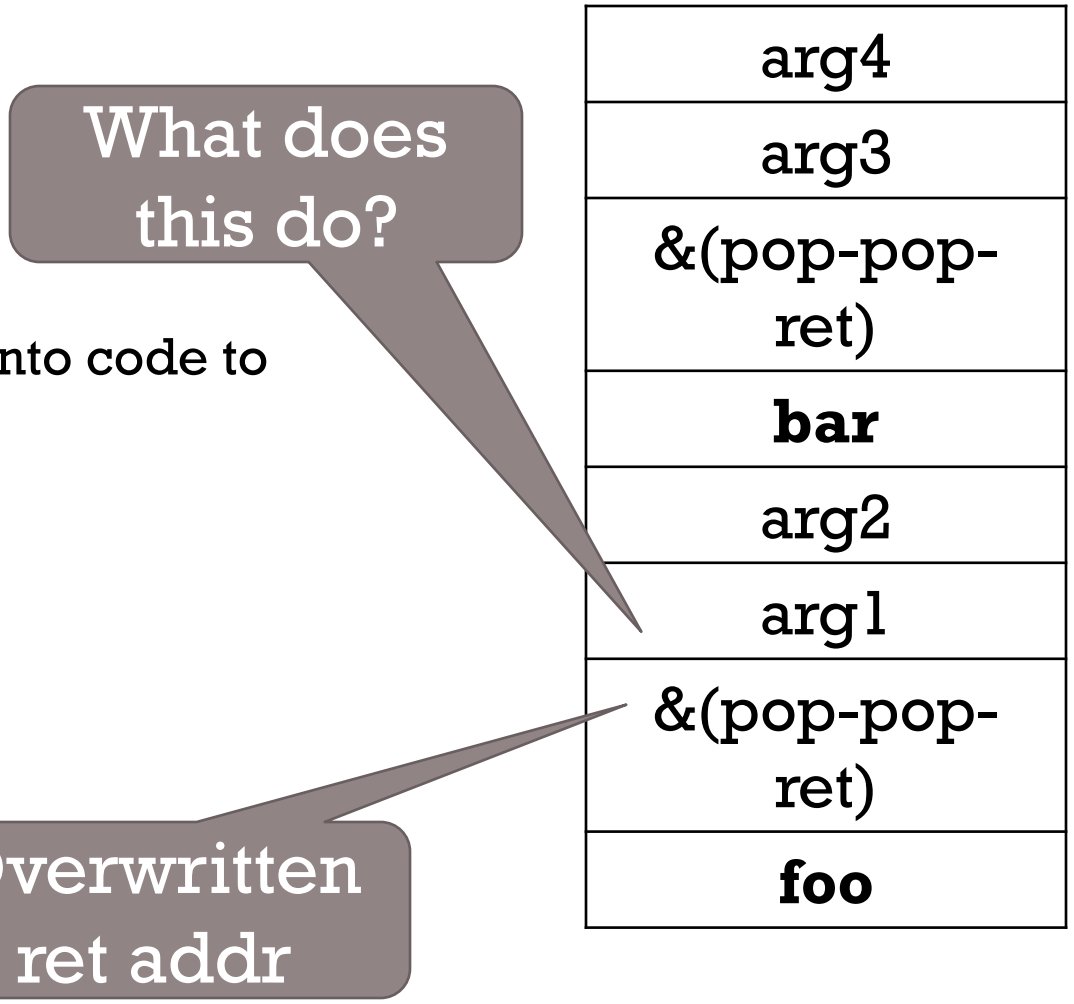


RETURN CHAINING

Suppose we want to call 2 functions in our exploit:

foo(arg1, arg2)
bar(arg3, arg4)

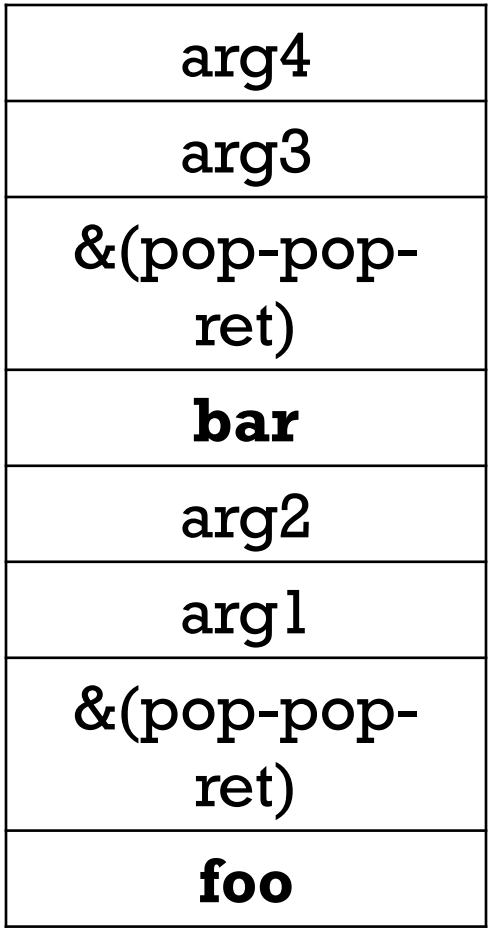
- Stack unwinds up
- First function returns into code to advance stack pointer
  - e.g., pop; pop; ret



RETURN CHAINING

- When **foo** is executing, &pop-pop-ret is at the saved EIP slot.

- When **foo** returns, it executes pop-pop-ret to clear up arg1 (pop), arg2 (pop), and transfer control to **bar** (ret)



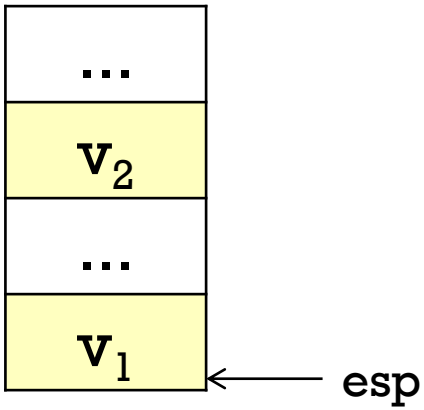
THERE ARE MANY *SEMANTICALLY EQUIVALENT* WAYS TO ACHIEVE THE SAME NET SHELLCODE EFFECT

Let's practice thinking in gadgets

AN EXAMPLE OPERATION

```
Mem[v2] = v1
```

Desired Logic



Stack

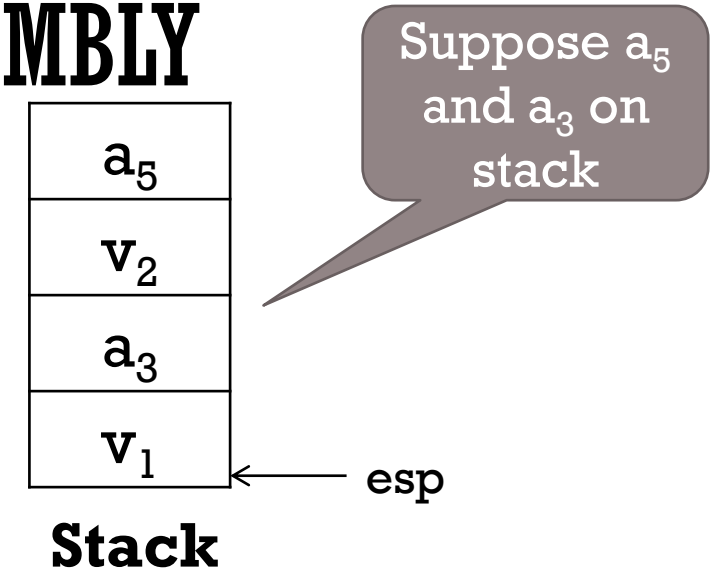
```
a1: mov eax, [esp] ; eax has v1
a2: mov ebx, [esp+8] ; ebx has v2
a3: mov [ebx], eax ; Mem[v2] = eax
```

Implementation 1

IMPLEMENTING WITH ASSEMBLY

```
Mem[v2] = v1
```

Desired Logic



eax	v_1
ebx	
eip	a_1

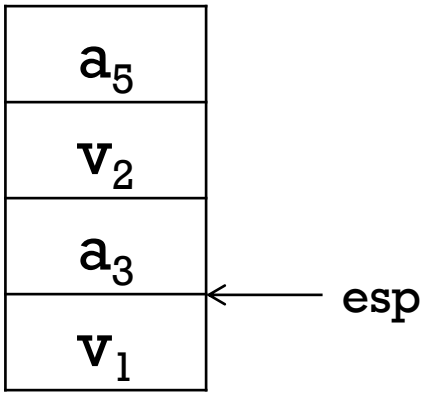
```
a1: pop eax
a2: ret
a3: pop ebx
a4: ret
a5: mov [ebx], eax
```

Implementation 2

IMPLEMENTING WITH ASSEMBLY

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	
eip	a_3

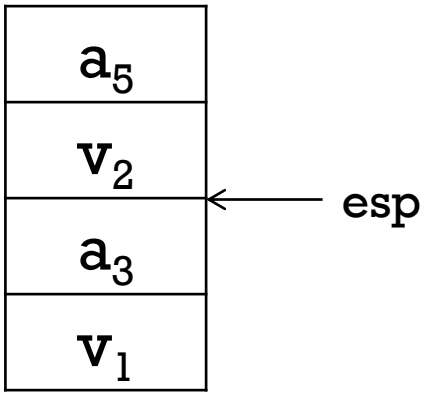
```
a1: pop eax
a2: ret
a3: pop ebx
a4: ret
a5: mov [ebx], eax
```

Implementation 2

IMPLEMENTING WITH ASSEMBLY

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_3

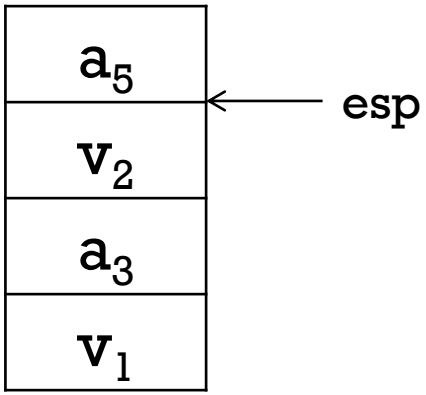
```
a1: pop eax
a2: ret
a3: pop ebx
a4: ret
a5: mov [ebx], eax
```

Implementation 2

IMPLEMENTING WITH ASSEMBLY

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_5

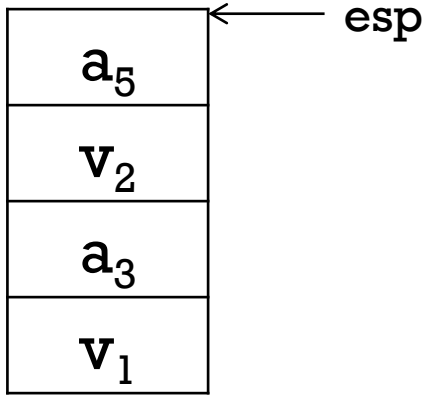
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

IMPLEMENTING WITH GADGETS!

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_5

```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Gadget 1

Gadget 2

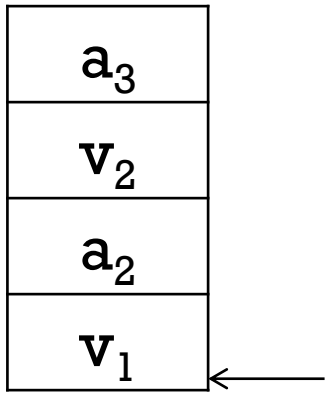
Implementation 2

EQUIVALENCE

```
Mem[v2] = v1
```

Desired Logic

semantically equivalent



Stack

"Gadgets"

```
a1: mov eax, [esp]
a2: mov ebx, [esp+8]
a3: mov [ebx], eax
```

Implementation 1

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

Implementation 2

GADGETS

- A gadget is a set of instructions for carrying out a semantic action
  - mov, add, etc.
- Gadgets typically have a number of instructions
  - One instruction = native instruction set
  - More instructions = synthesize <- ROP
- Gadgets in ROP generally (but not always) end in return

RETURN-ORIENTED PROGRAMMING (ROP)

`Mem[v2] = v1`

Desired *Shellcode*

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

Desired store executed!

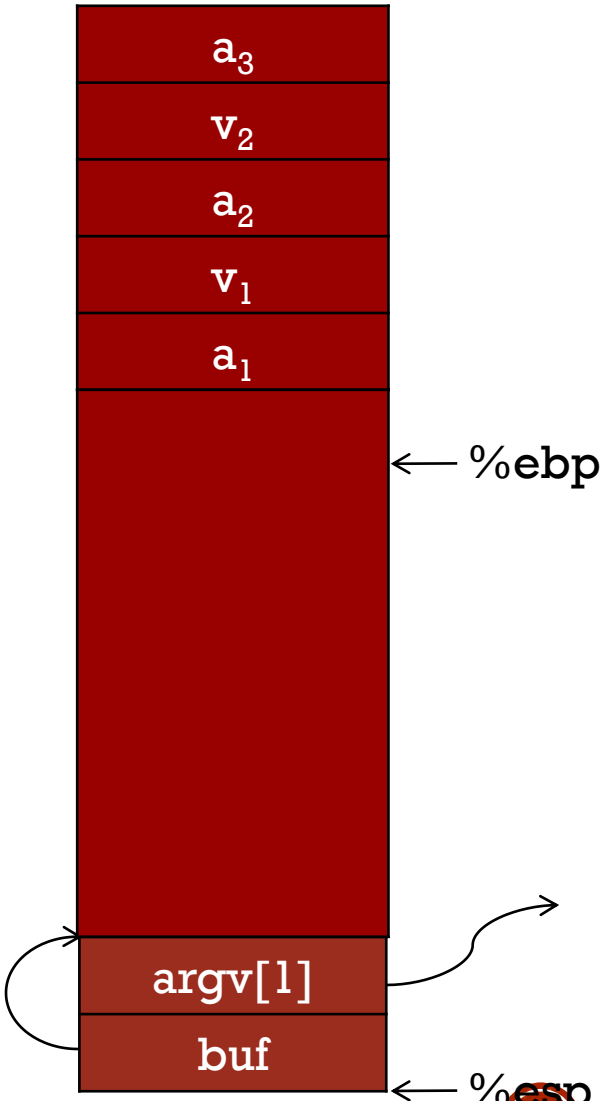


Image by Dino Dai Zovi

RO(P?) PROGRAMMING

1. Disassemble code
2. Identify *useful* code sequences as gadgets
3. Assemble gadgets into desired shellcode