Host Vulnerabilities

UT CS361S

SPRING 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

Today: ***very brief*** overview of ***Control Flow Hijacking***

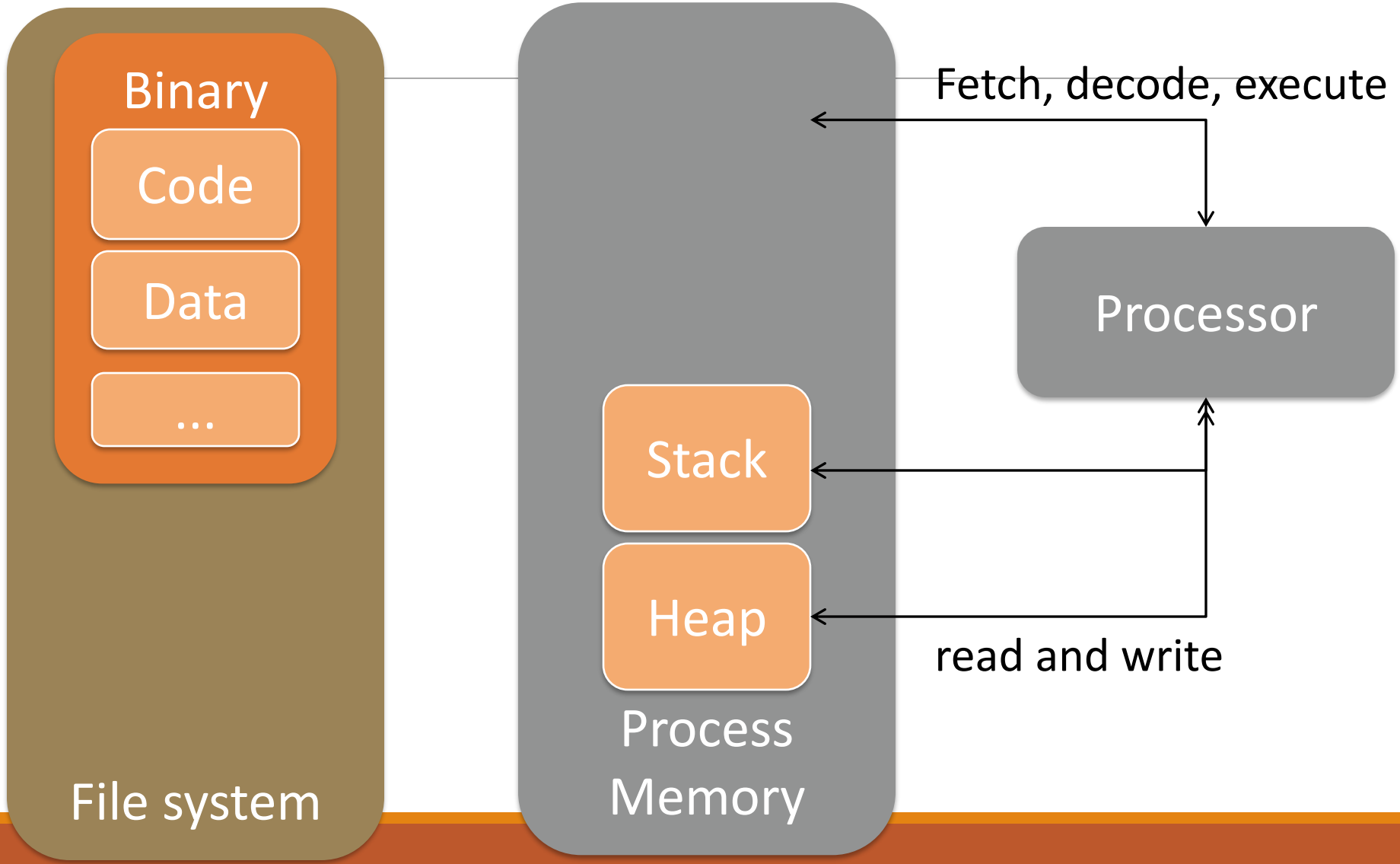
- There are other types of vulnerabilities (e.g misconfigured)
- Control Flow Hijacking is probably the hardest to grasp

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

stdcall (callee)

```
    push arg1
    push arg2
    push arg3
    call proc

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

cdecl (caller)

```
    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

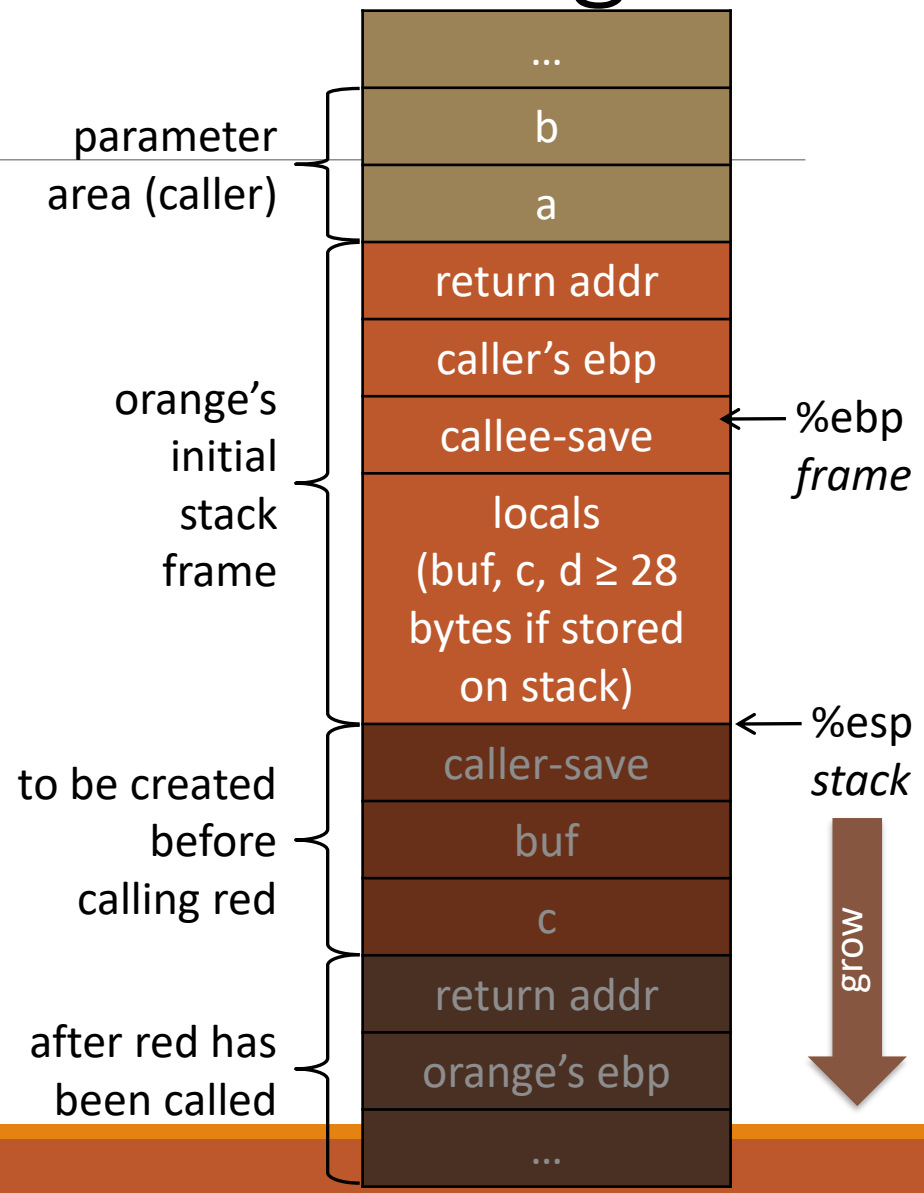
ESP

- 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

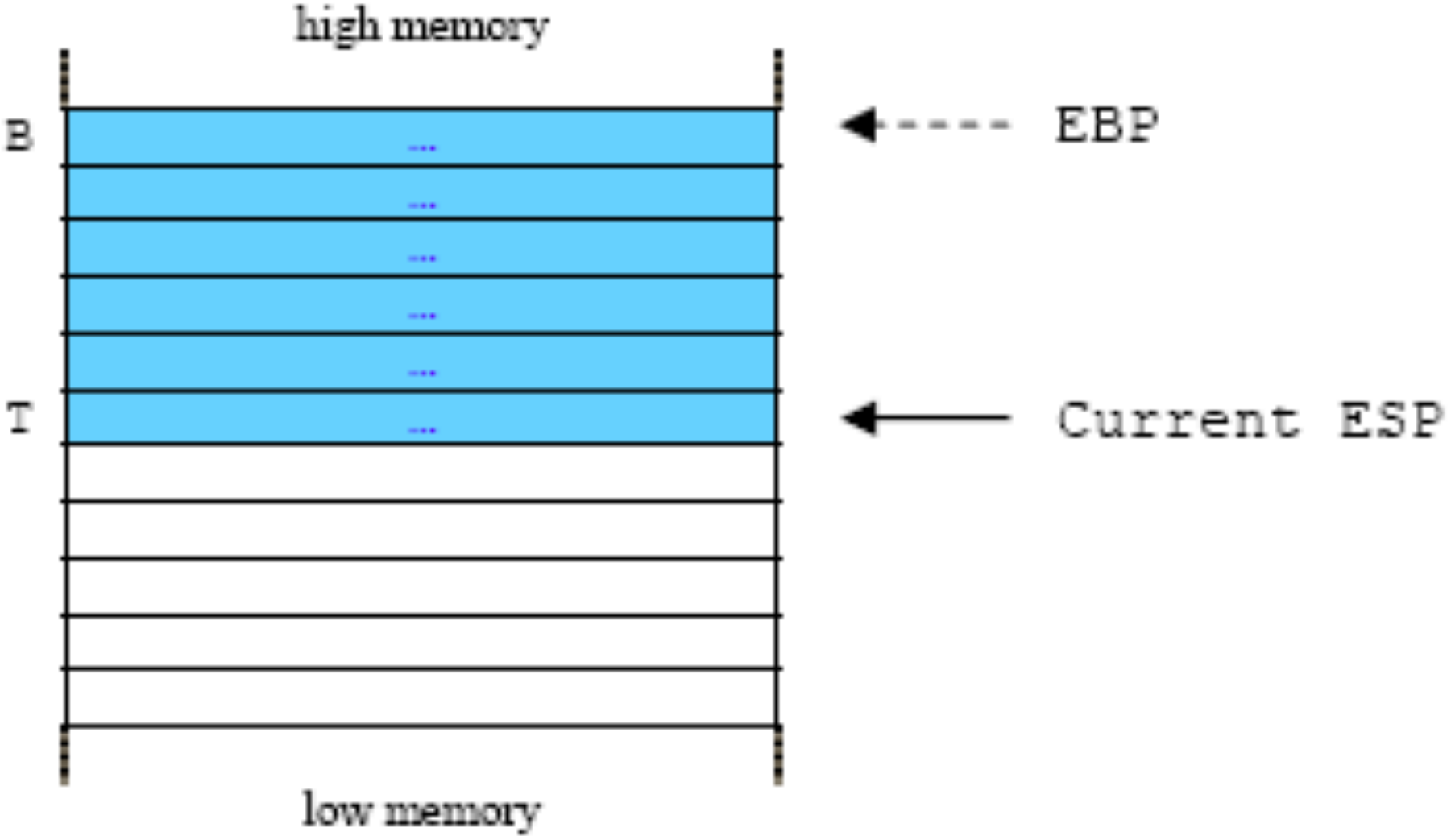
GDB Walkthrough – C Code

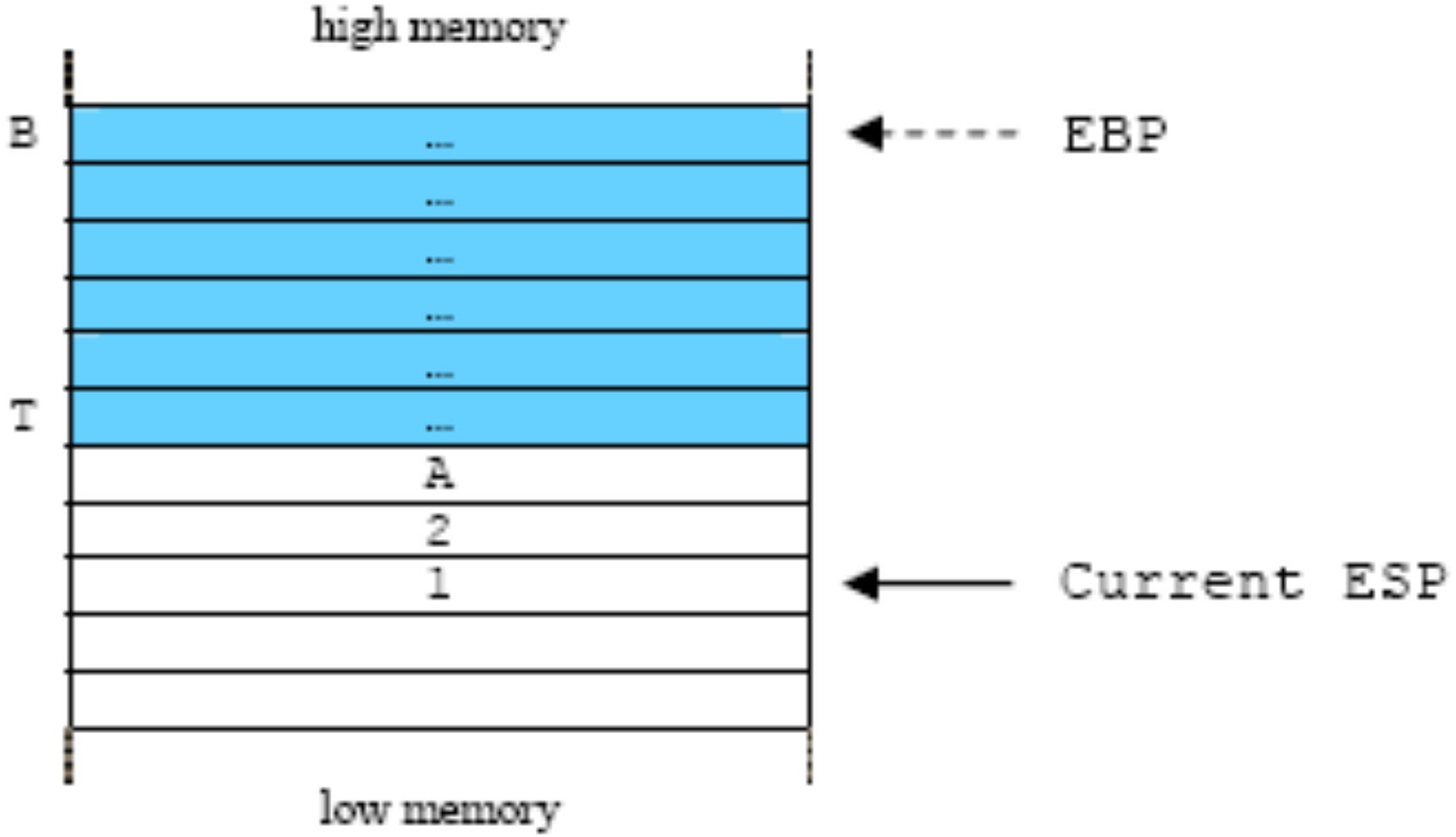
```c
#include <stdio.h>

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;
}
```

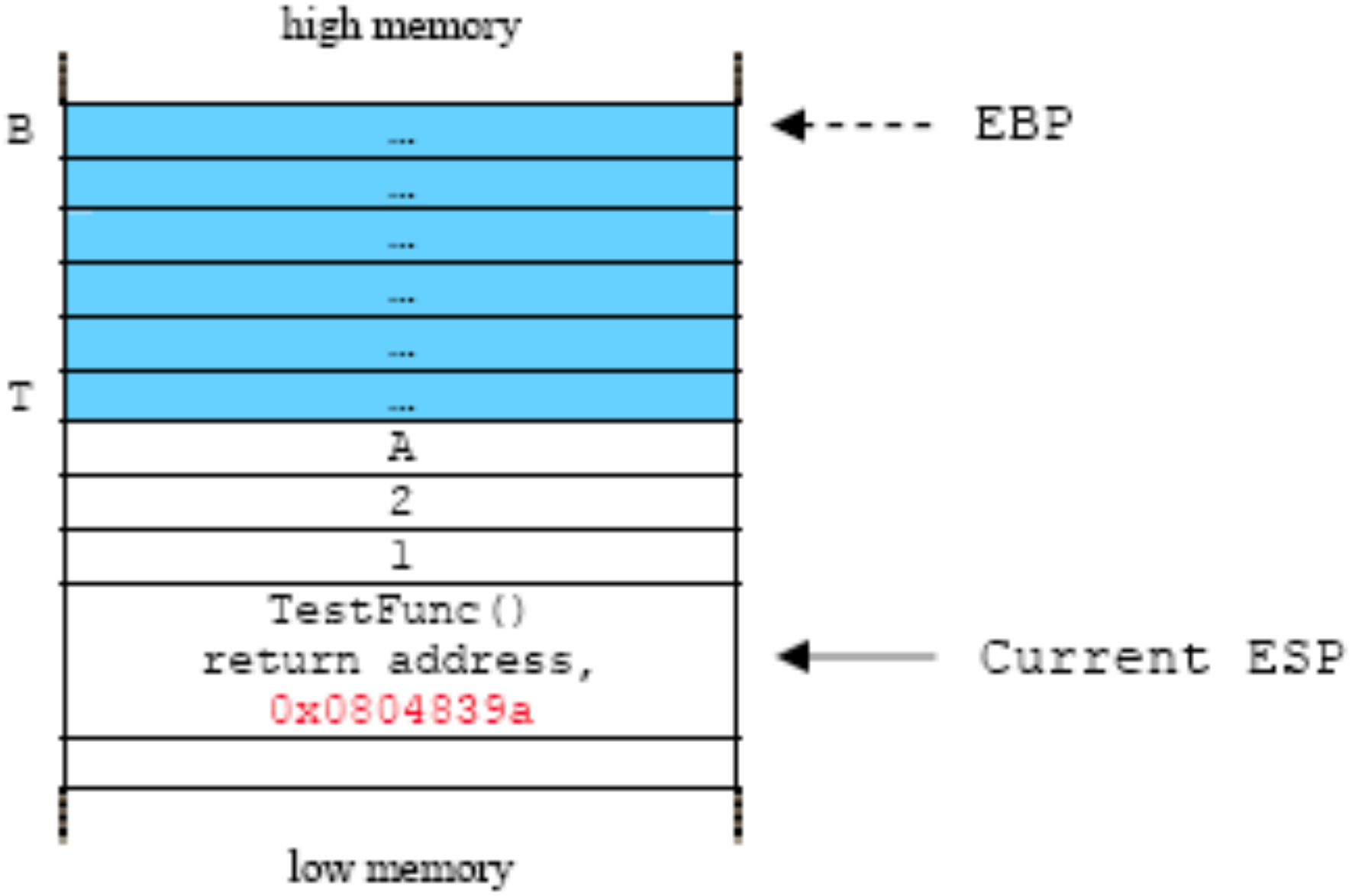
GDB Walkthrough – Call TestFunc

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

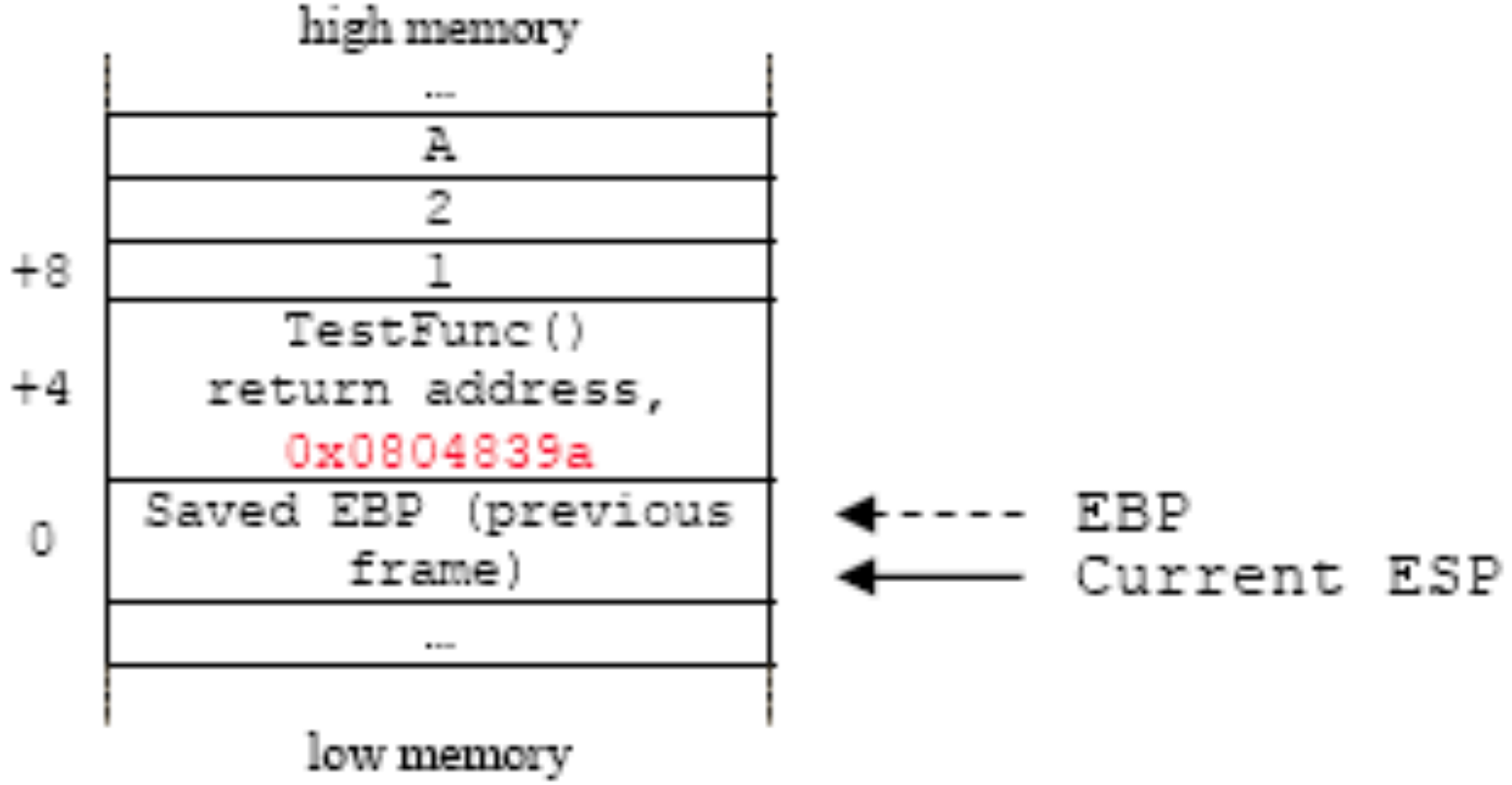
GDB Walkthrough – TestFunc() C Code

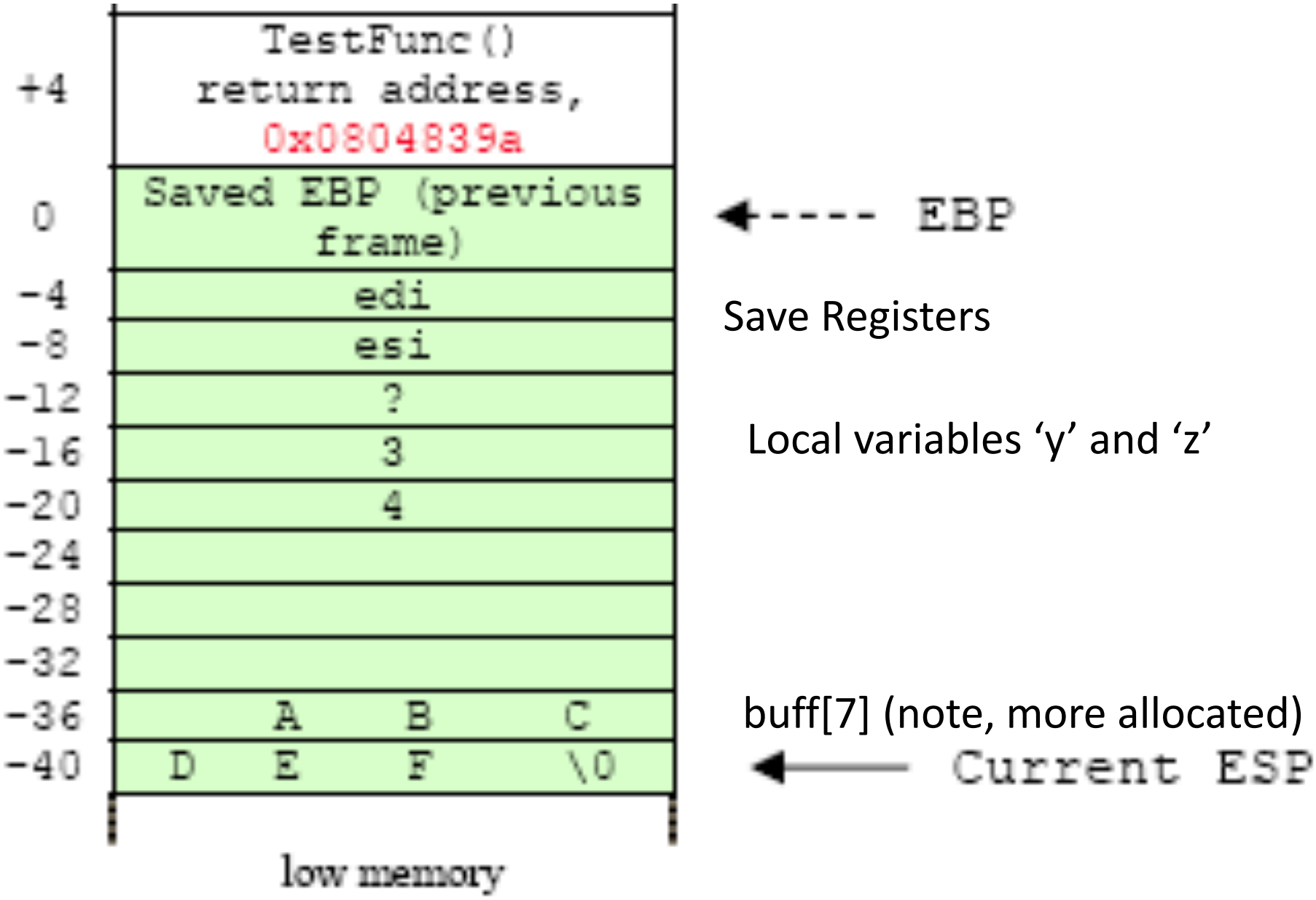
```
int TestFunc(int parameter1,int parameter2,char parameter3)
{
int y = 3, z = 4;
char buff[7] = "ABCDEF";

// 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]
```

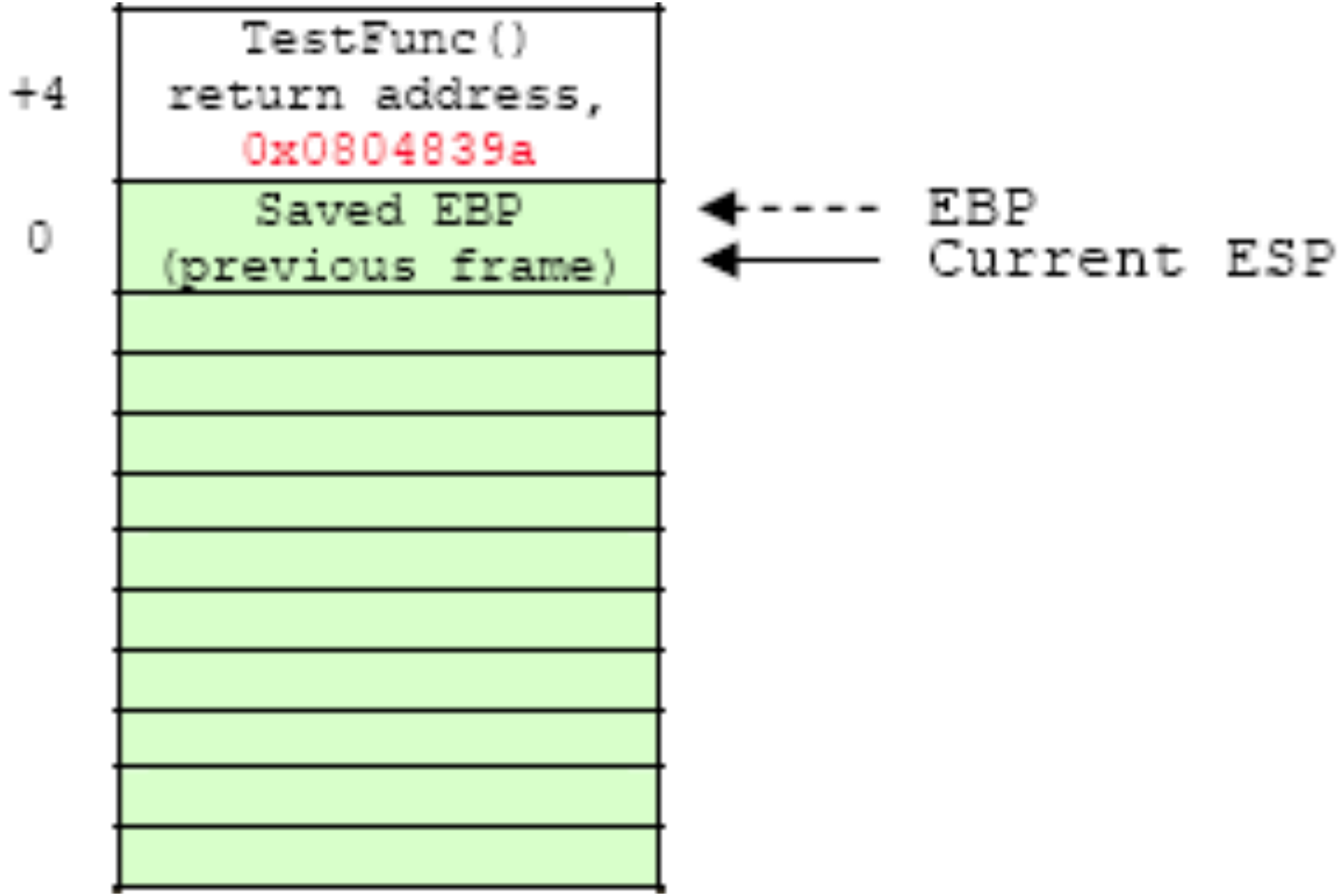
32 bytes allocated on stack (0x20). Variables Loaded into this space (not shown).


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 allocated)
-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]
```

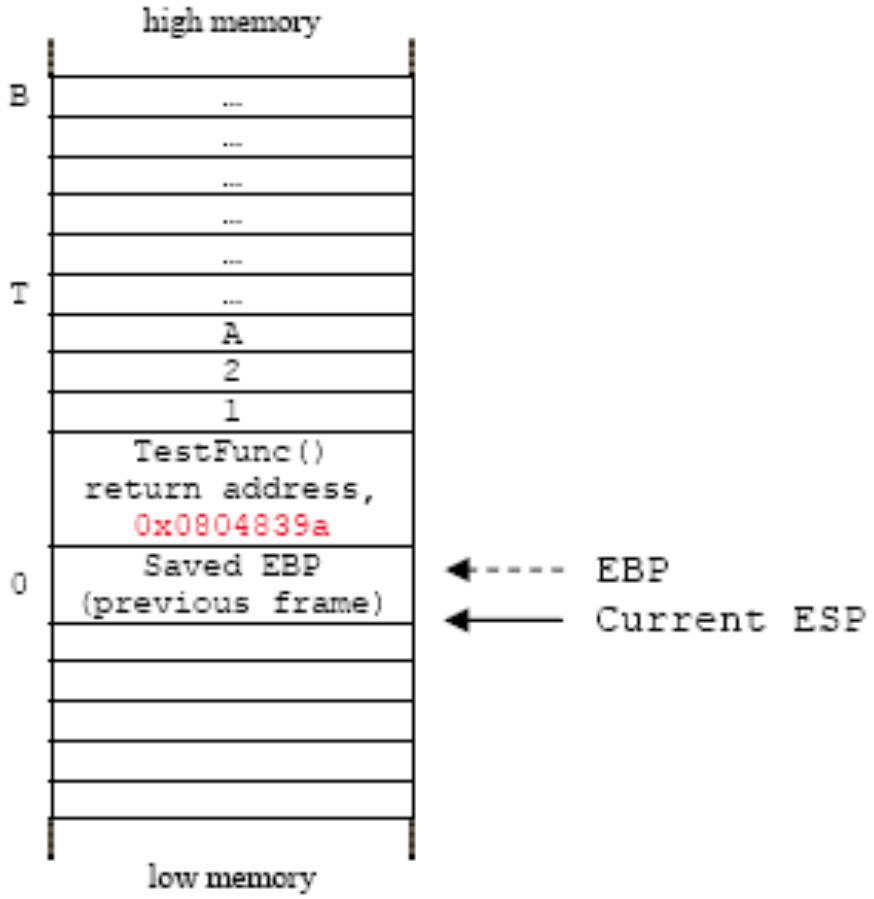

TestFunc()
+4
return address,
0x0804839a
Saved EBP
0
(previous frame)
EBP
Current ESP

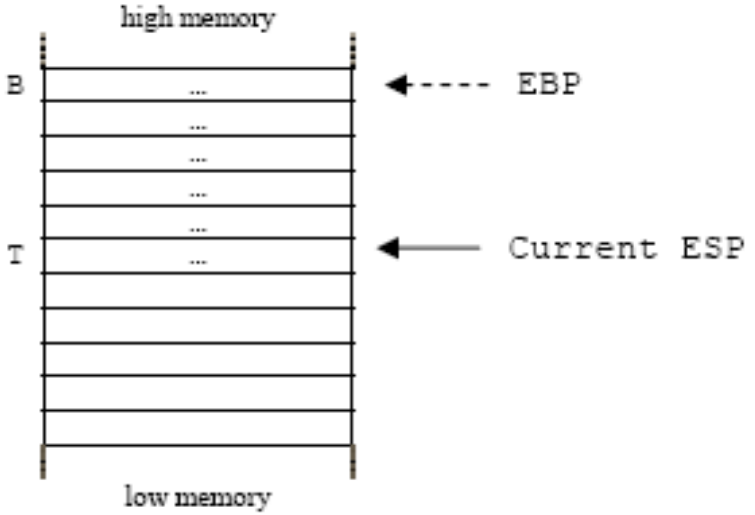
GDB Walkthrough – TestFunc() Exit, part 2

```
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
Saved EBP
(previous frame)
0
EBP
Current ESP
low memory


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

What are Buffer Overflows?

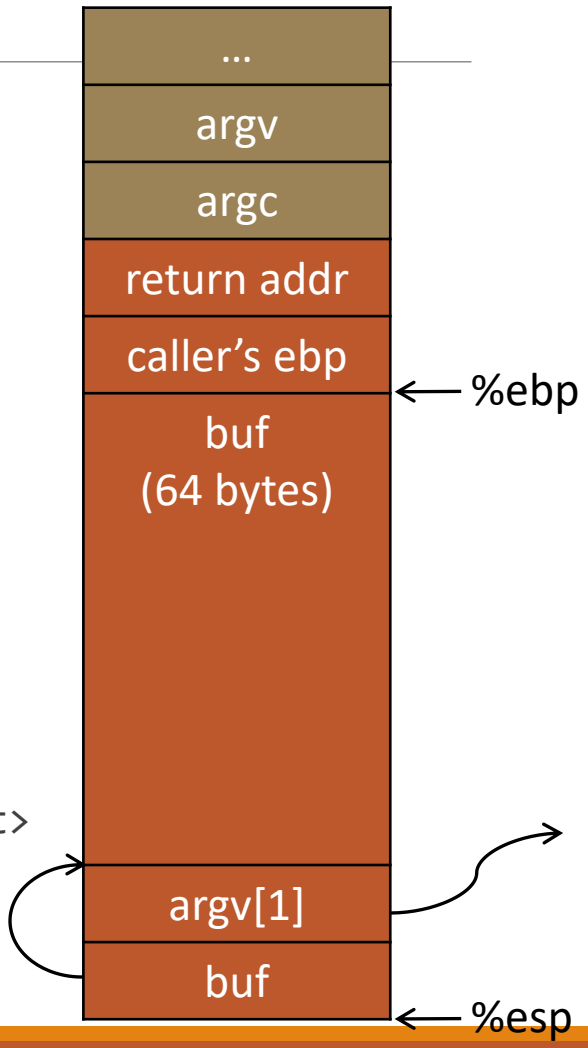
A ***buffer overflow*** occurs when data is written outside of the space allocated for the buffer.

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

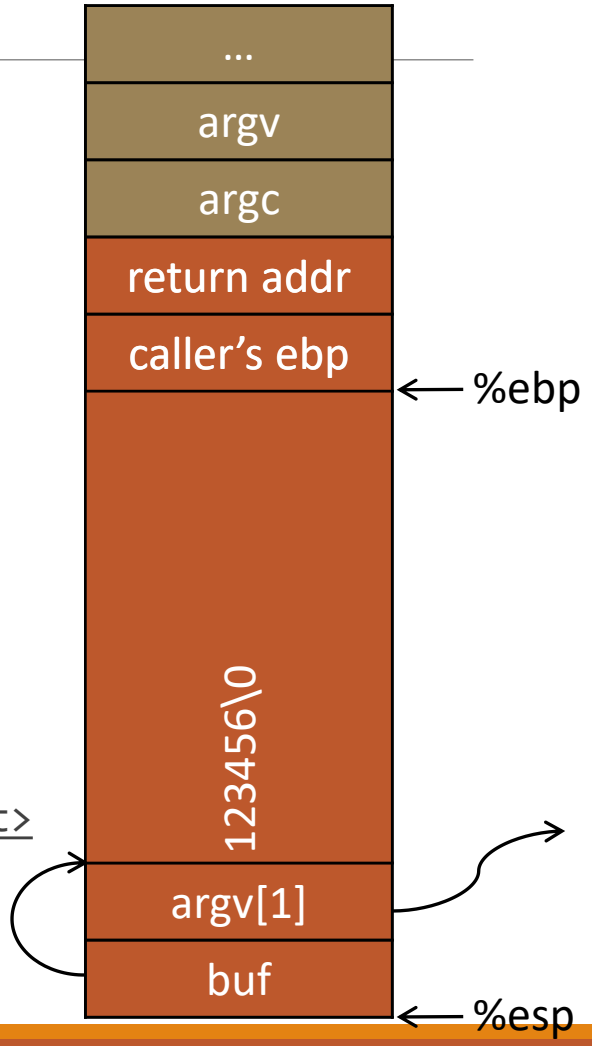
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

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
```



27

“123456”

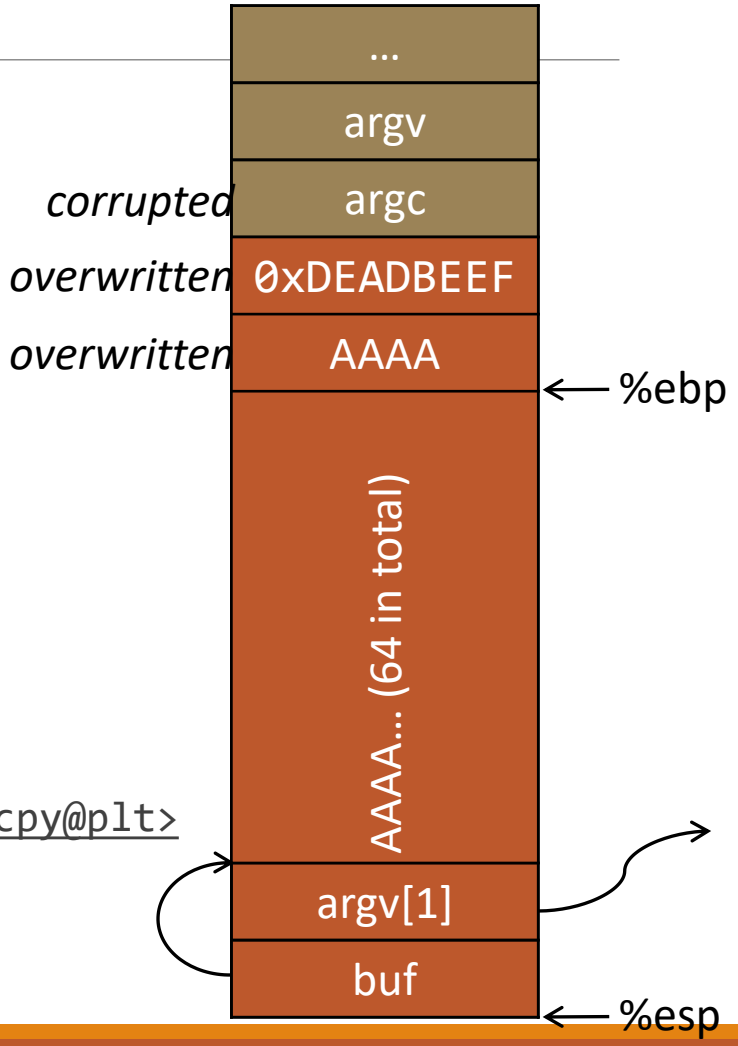
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

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
```



28

“A”x68 . “\xEF\xBE\xAD\xDE”

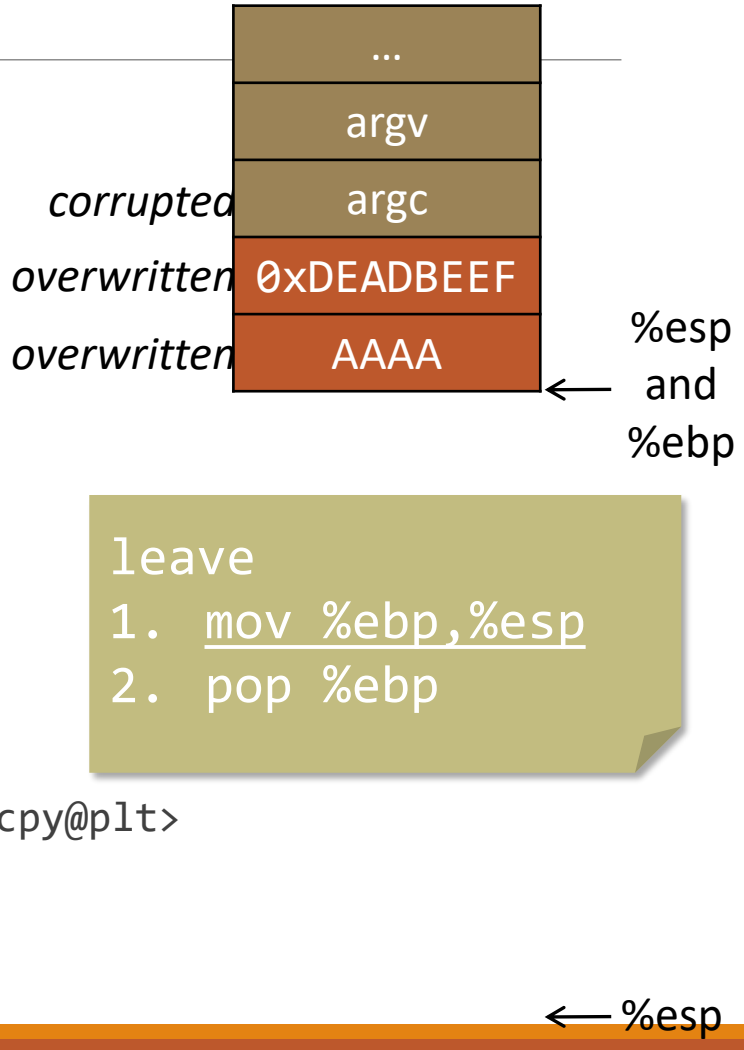
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

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
```



29

Frame teardown—1

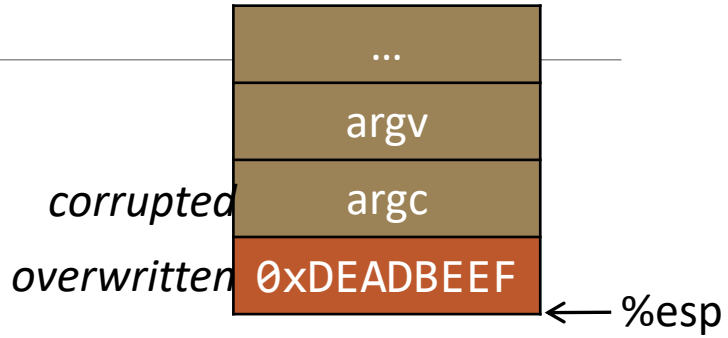
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

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@plt>
=> 0x080483ff <+27>:    leave
   0x08048400 <+28>:    ret
```



30

Frame teardown—2

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



%ebp = AAAA

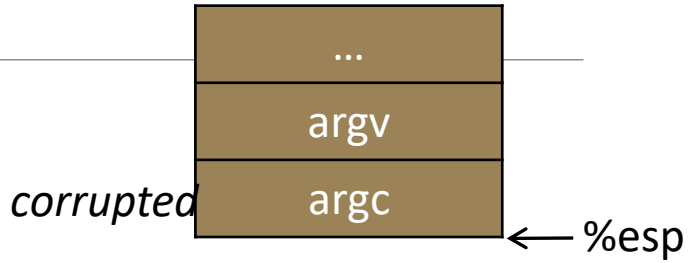
```
Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $72,%esp
   0x080483ea <+6>:     mov    12(%ebp),%eax
   0x080483ed <+9>:     mov    4(%eax),%eax
   0x080483f0 <+12>:    mov    %eax,4(%esp)
   0x080483f4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

```
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)

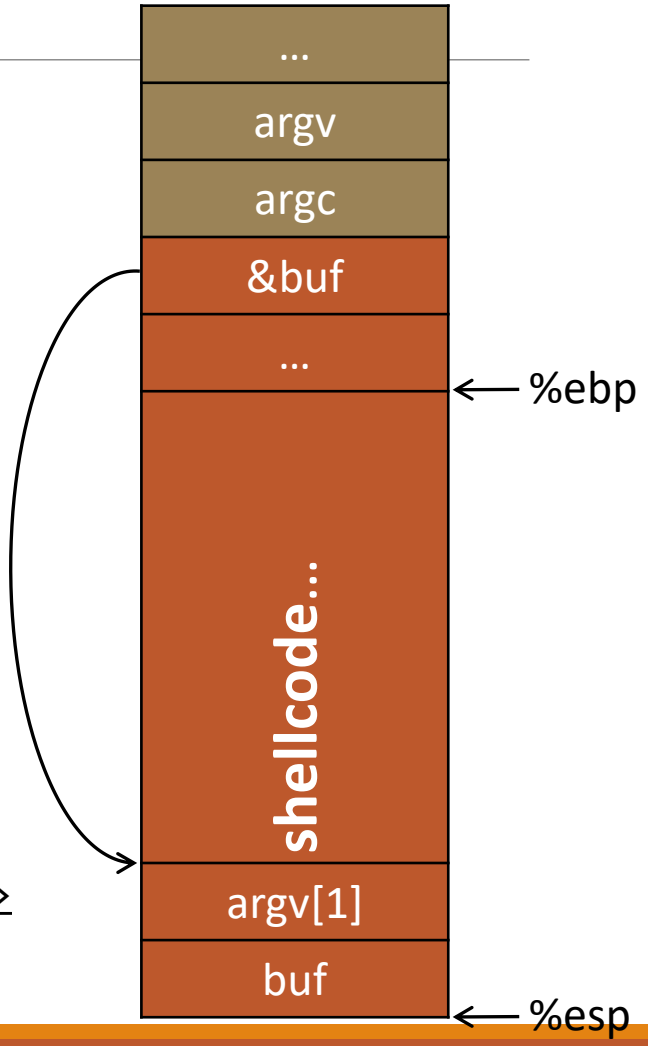
32

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

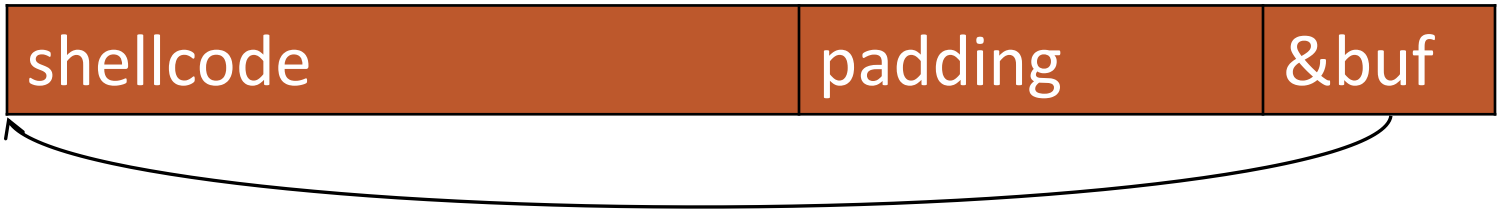



```
…
0x080483fa <+22>: call   0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

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***