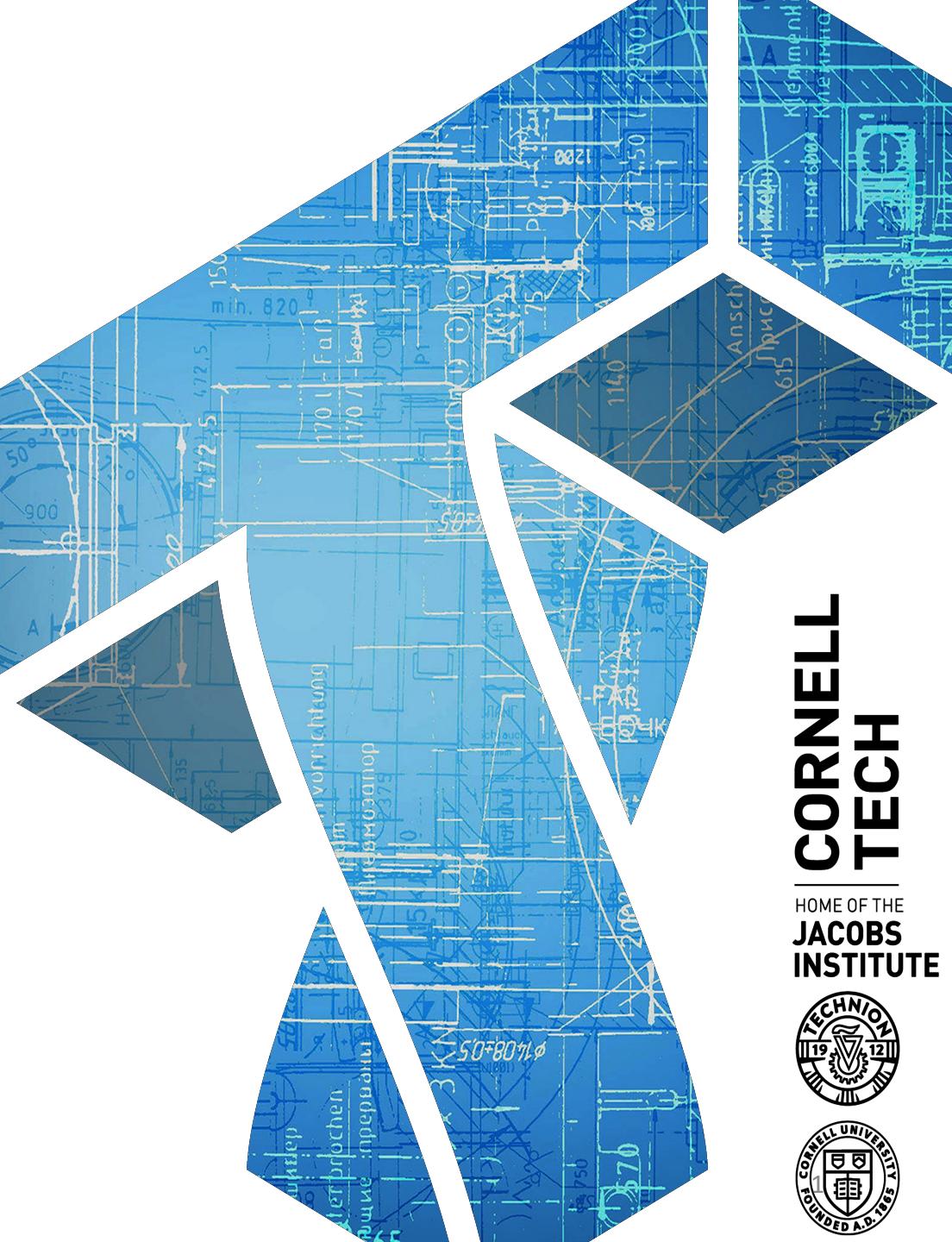


CS 5435: Buffer overflows

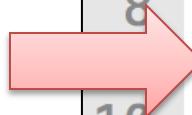
Instructor: Tom Ristenpart

<https://github.com/tomrist/cs5435-spring2024>



Running demo example (modified from Gray hat hacking book linked in resources)

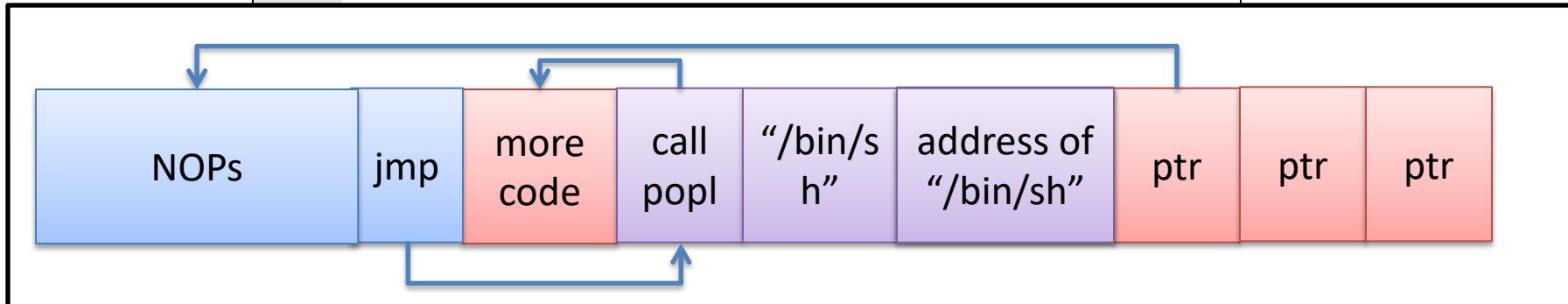
```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* temp1 )
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
12
13
14 int main(int argc, char* argv[] )
15 {
16     greeting( argv[1] );
17     printf( "Bye %s\n", argv[1] );
18 }
```



Small buffers

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* t
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
```

What if 400 is changed to a small value, say 10?



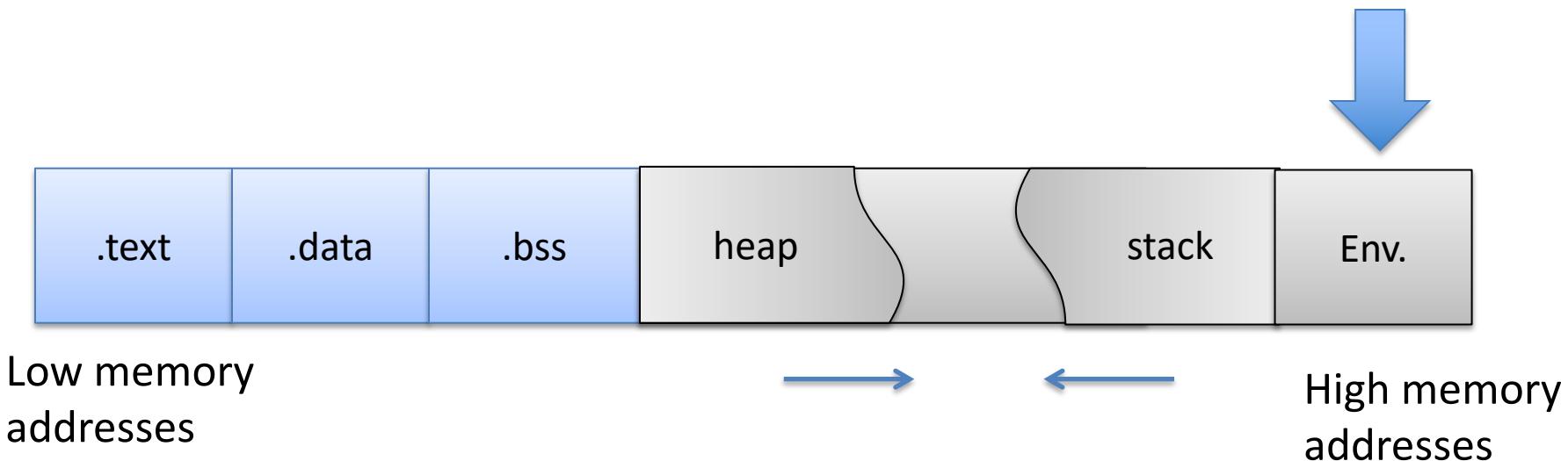
Small buffers

Use an environment variable to store exploit buffer

`execve("meet", argv, envp)`

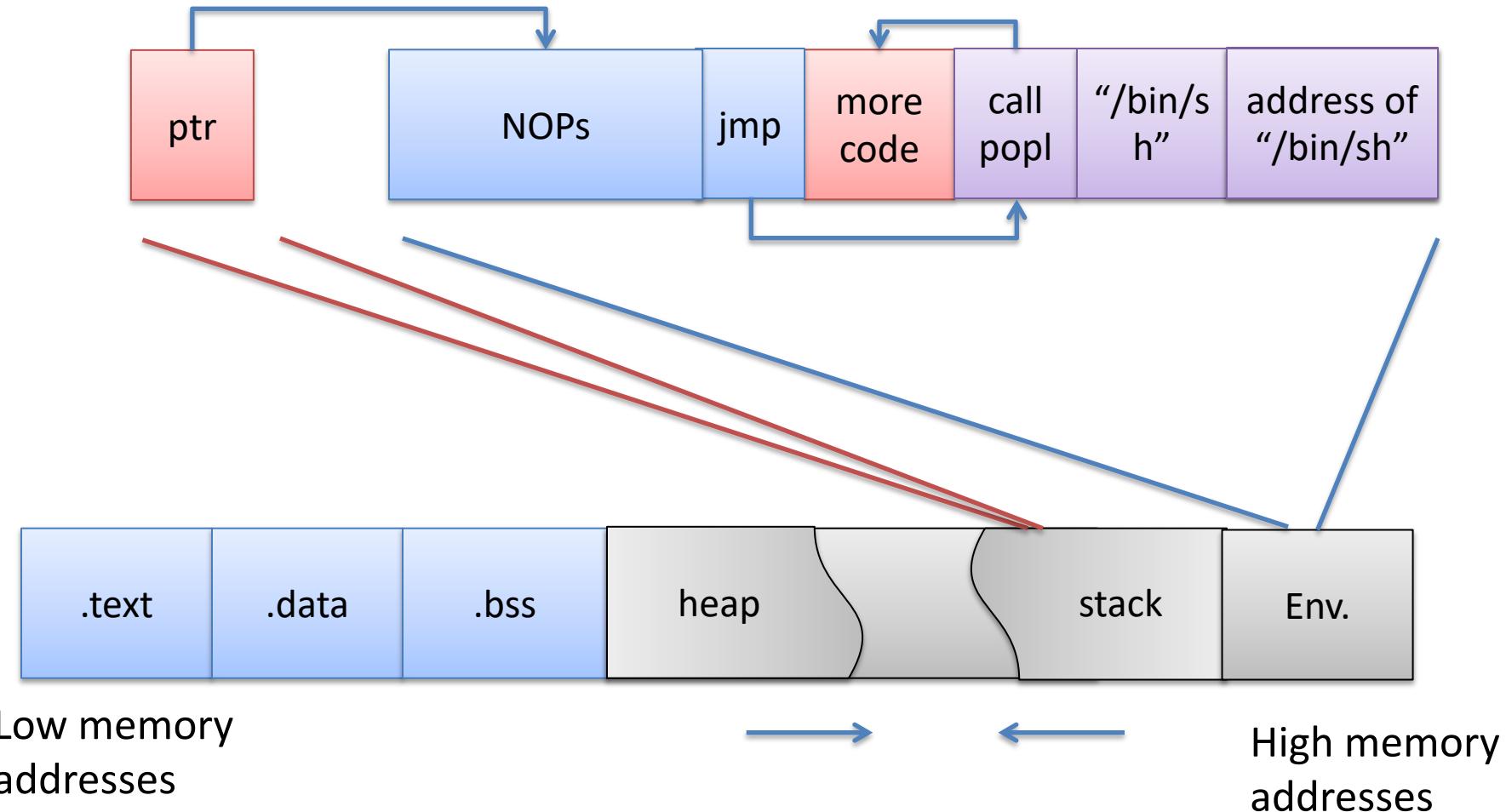
`envp` = array of pointers to strings (just like `argv`)

- Normally, bash passes in this array from your shell's environment
 - You can also pass it in explicitly via execve()



Small buffers

Return address overwritten with ptr to environment variable



There are other ways to inject code

- examples: .dtors (Gray Hat book), function pointers, ...
- dig around in Phrack articles ...
- And ways to avoid injection entirely:
 - Return-into-libc
 - Return-oriented programming
 - Reuse existing code in process memory maliciously

Integer overflows

```
void func(int a, char v)
    char buf[128];
    init(buf);
    buf[a] = v;
}
```

`&buf[a]` could be return address

Integer overflows

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

int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3){
        return -1;
    }
```

```
nova:signed {100} ./width1 5 hello
s = 5
hello
nova:signed {101} ./width1 80 hello
Oh no you don't!
nova:signed {102} ./width1 65536 hello
s = 0
Segmentation fault (core dumped)
```

```
i = atoi(argv[1]);
s = i;

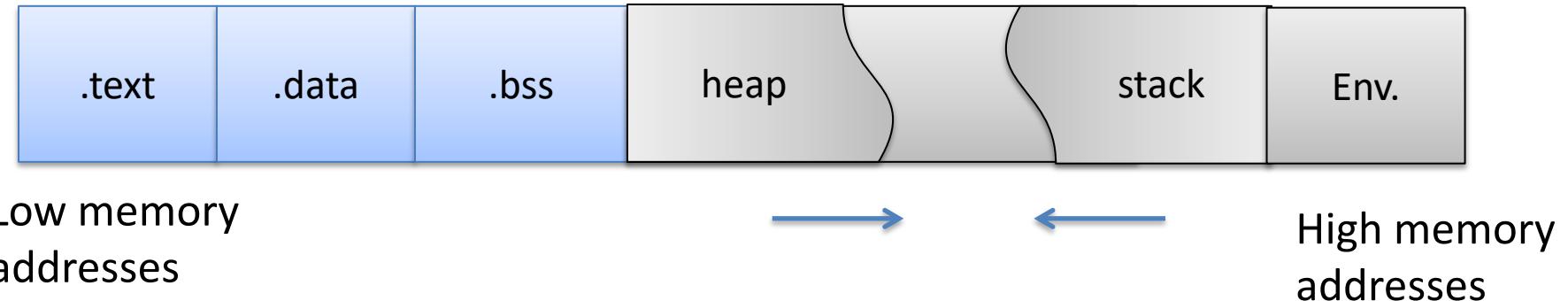
if(s >= 80) { /* [w1] */
    printf("Oh no you don't!\n");
    return -1;
}

printf("s = %d\n", s);

memcpy(buf, argv[2], i);
buf[i] = '\0';
printf("%s\n", buf);

return 0;
```

Heap overflows



Format-string vulnerabilities

```
printf( const char* format, ... )
```

```
printf( "Hi %s %s", argv[0], argv[1] )
```

```
void main(int argc, char* argv[])
{
    printf( argv[1] );
}
```

argv[1] = "%s%s%s%s%s%s%s%s%s%s%s"

Attacker controls format string gives all sorts of control

Can do control hijacking directly

	<i>Buffer Overflow</i>	<i>Format String</i>
public since	mid 1980's	June 1999
danger realized	1990's	June 2000
number of exploits	a few thousand	a few dozen
considered as	security threat	programming bug
techniques	evolved and advanced	basic techniques
visibility	sometimes very difficult to spot	easy to find

From “Exploiting format string vulnerabilities”

Bad C library functions

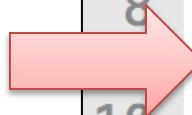
- strcpy
- strcat
- scanf
- gets
- “More” safe versions: strncpy, strncat, etc.
 - Specify max number of bytes to copy
 - These are not foolproof either!

Summary so far

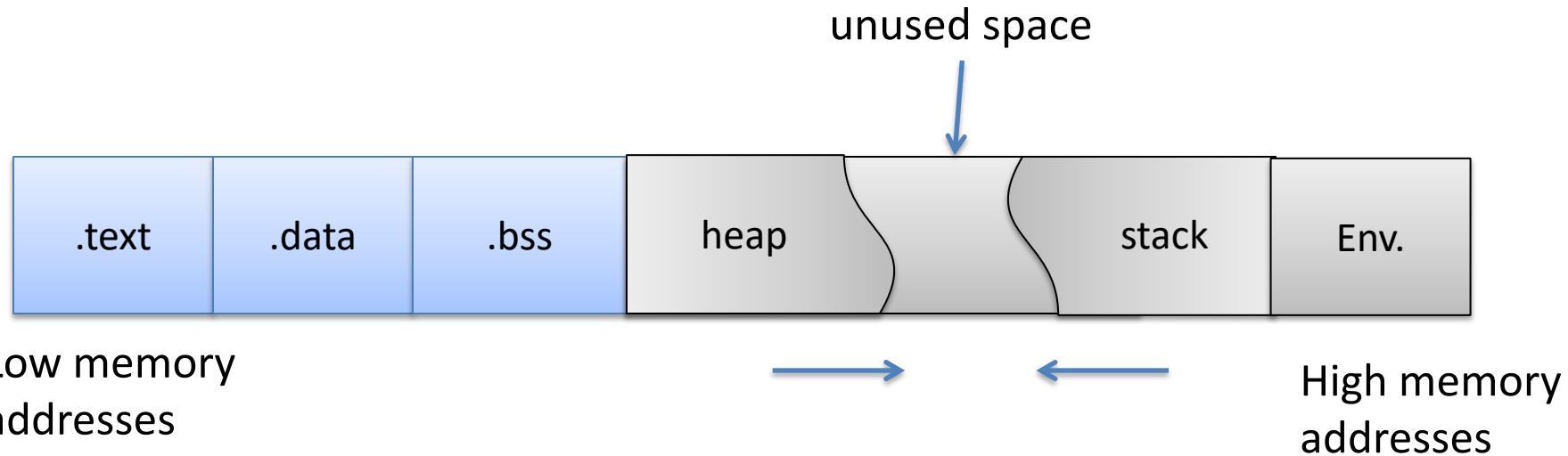
- Classic buffer overflow
 - corrupt program control data
 - hijack control flow
- Integer overflow, signedness, format string, heap overflow, ...
- These were all local privilege escalation vulns
 - Similar concepts for remote vulnerabilities
- Defenses?

Running demo example (modified from Gray hat hacking book linked in resources)

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* temp1 )
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
12
13
14 int main(int argc, char* argv[] )
15 {
16     greeting( argv[1] );
17     printf( "Bye %s\n", argv[1] );
18 }
```



Process memory layout



.text:
machine code of executable

.data:
global initialized variables

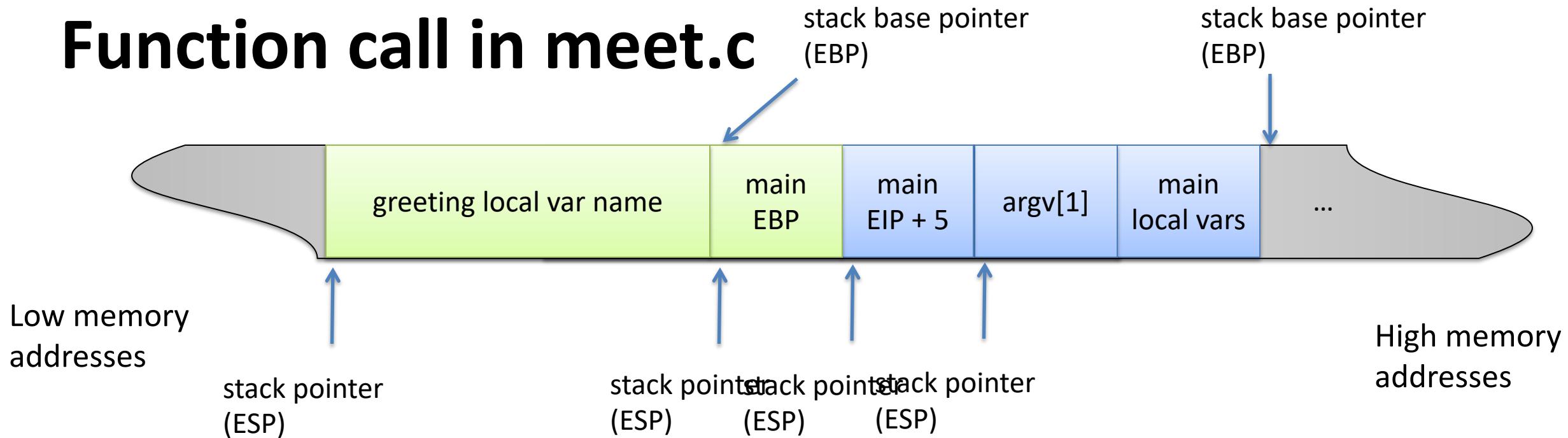
.bss:
“below stack section”
global uninitialized variables

heap:
dynamic variables

stack:
local variables, track func calls

Env:
environment variables,
arguments to program

Function call in meet.c



```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
0x080484b3 <+0>: push %ebp
0x080484b4 <+1>: mov %esp,%ebp
0x080484b5 <+2>: mov 0xc(%ebp),%eax
0x080484b6 <+3>: add $0x4,%eax
0x080484b7 <+4>: mov (%eax),%eax
0x080484b8 <+5>: push %eax
0x080484bf <+12> eip call 0x804846b <greeting>
0x080484c4 <+17> eip add $0x4,%esp
```

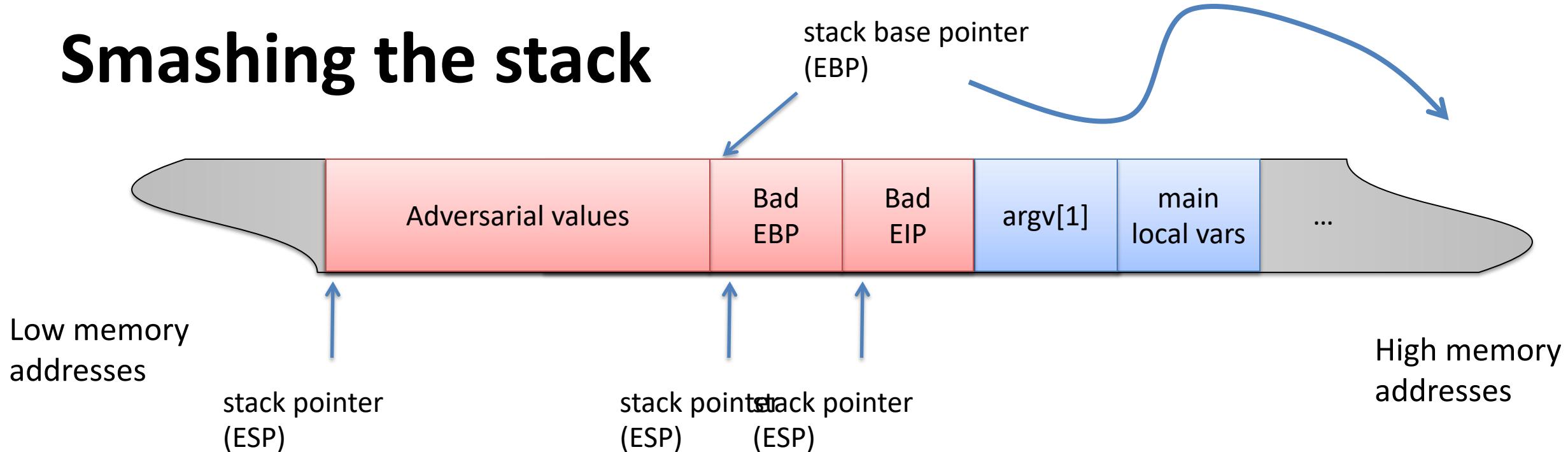
Pushing argv[1] onto stack

```
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x0804846b <+0> eip push %ebp
0x0804846c <+1> eip mov %esp,%ebp
0x0804846d <+2> eip sub $0x190,%esp
```

.... (more stuff including strcpy) ...

```
0x080484b1 <+70> eip leave
0x080484b2 <+71> eip ret
```

Smashing the stack



```
student@5435-hw4-vm:~/demo$ gdb -q meet
Reading symbols from meet...done.
(gdb) disassemble main
Dump of assembler code for function main:
0x080484b3 <+0>: push %ebp
0x080484b4 <+1>: mov %esp,%ebp
0x080484b5 <+2>: mov 0xc(%ebp),%eax
0x080484b6 <+3>: add $0x4,%eax
0x080484b7 <+4>: mov (%eax),%eax
0x080484b8 <+5>: push %eax
0x080484b9 <+6>: call 0x804846b <greeting>
0x080484bf <+12>: add $0x4,%esp
0x080484c4 <+17>:
```

Pushing argv[1]
onto stack

Bad eip

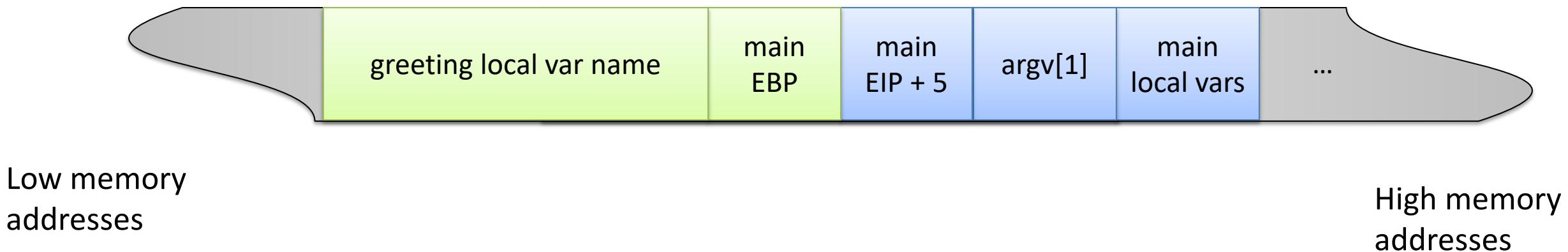
???

```
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x0804846b <+0>: push %ebp
0x0804846c <+1>: mov %esp,%ebp
0x0804846e <+3>: sub $0x190,%esp
```

.... (more stuff including strcpy) ...

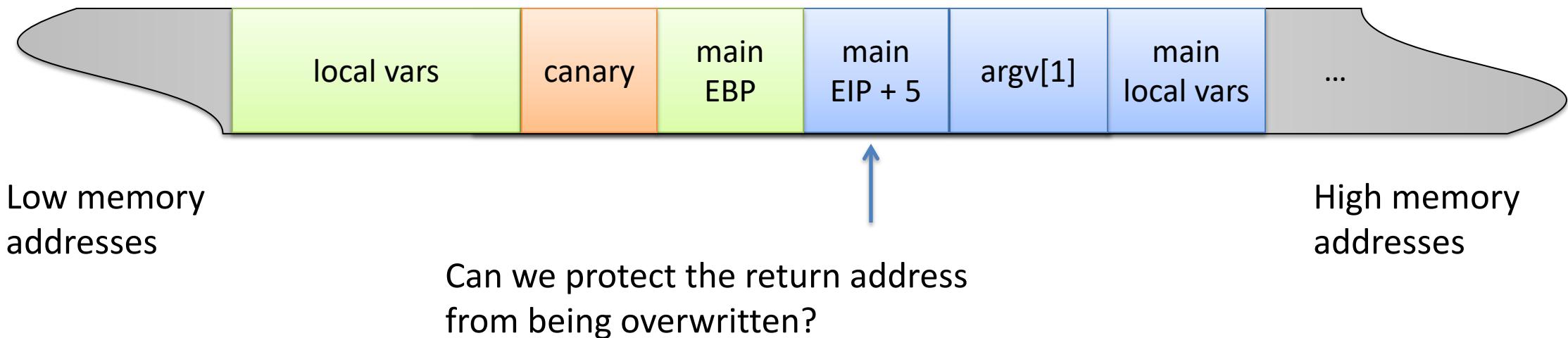
```
0x080484b1 <+70> eip leave
0x080484b2 <+71> eip ret
```

Countermeasures?



- Stack protections
- Address space layout randomization
- W^X (non-executable stack)
- Control-flow integrity

Detecting buffer overflows: stack canaries



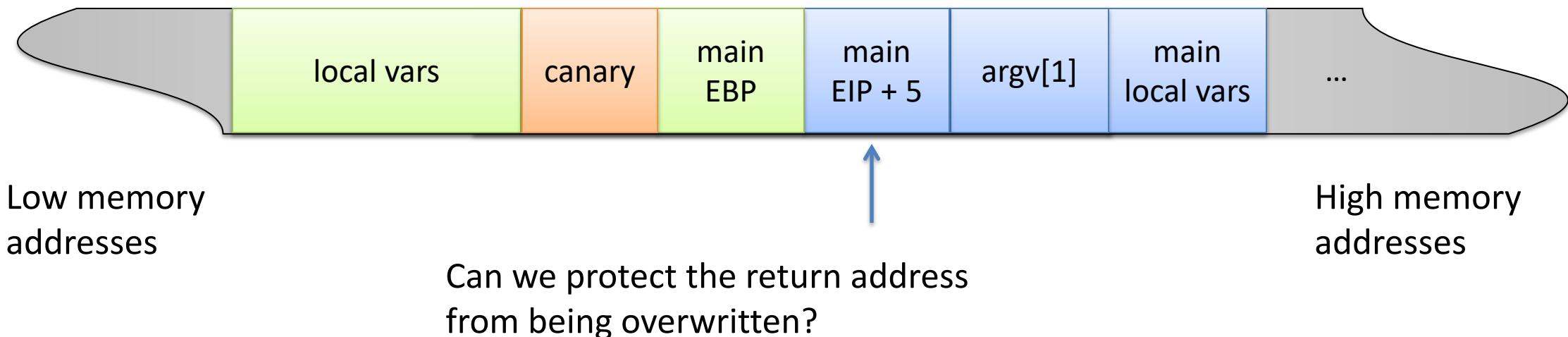
Before executing function, write canary value to stack

Three type:

- Random value (choose once for whole process)
 - Terminator canary (NULL bytes / EOF / etc. so string functions won't copy past canary)
 - Random XOR canary (XOR random value and stored EIP and/or EBP)

On end of function, check that canary is correct, if not terminate process

Detecting buffer overflows: stack canaries



StackGuard paper (1998):

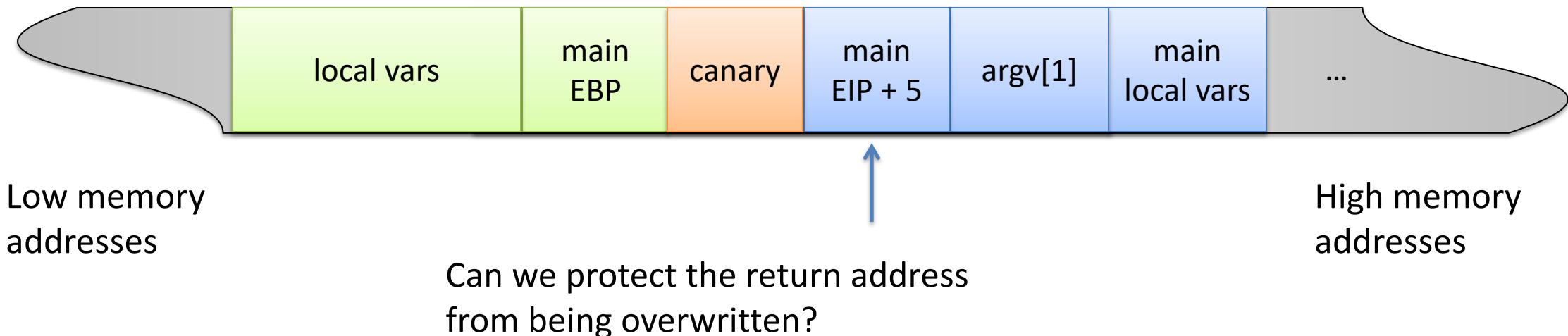
- GCC extension that adds runtime canary checking between saved EIP and EBP
 - 8% overhead on Apache

ProPolice (IBM gcc patches 2001-2005):

- Moved canary to after saved EBP
 - Rearranges local variables so arrays are highest in stack

What is the problem with this?

Detecting buffer overflows: stack canaries



StackGuard paper (1998):

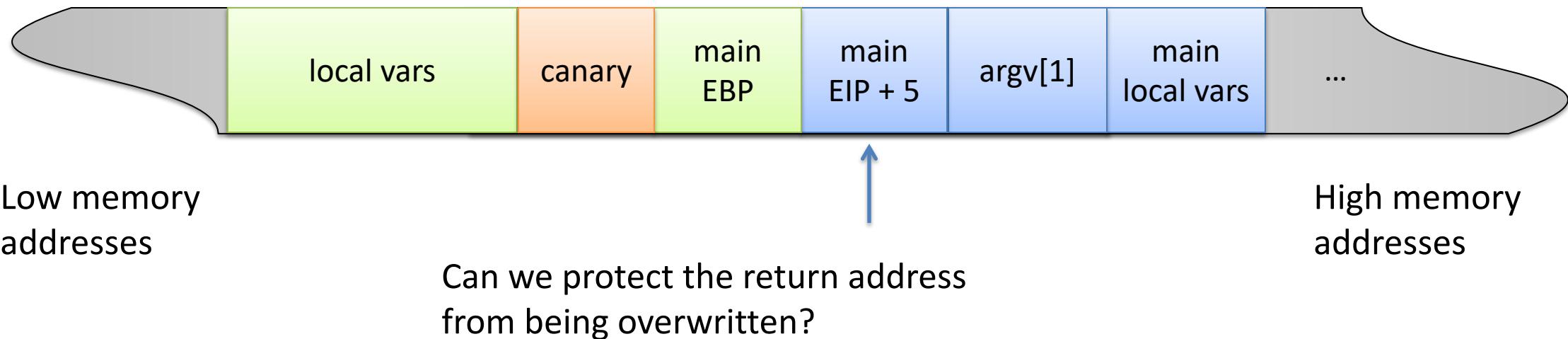
- GCC extension that adds runtime canary checking between saved EIP and EBP
 - 8% overhead on Apache

ProPolice (IBM gcc patches 2001-2005):

- Moved canary to after saved EBP
 - Rearranges local variables so arrays are highest in stack

What is the problem with this?

Detecting buffer overflows: stack canaries



StackGuard paper (1998):

- GCC extension that adds runtime canary checking between saved EIP and EBP
 - 8% overhead on Apache

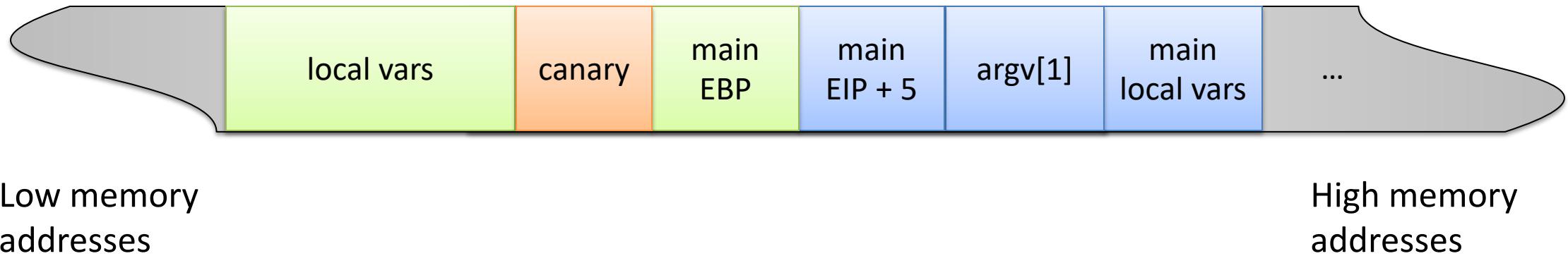
ProPolice (IBM gcc patches 2001-2005):

What is the problem with this?

- Moved canary to after saved EBP
- Rearranges local variables so arrays are highest in stack

Protect local var pointers

Detecting buffer overflows: stack canaries



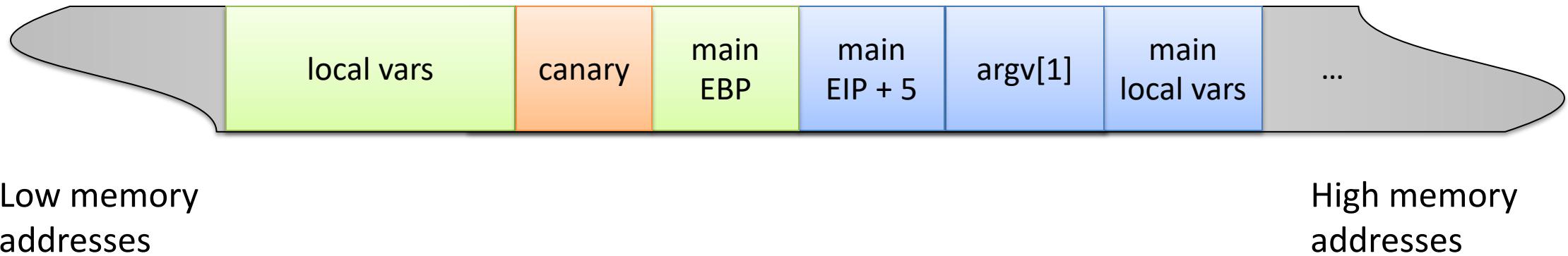
Low memory
addresses

High memory
addresses

Contemporary gcc implementation

Flag	Default?	Notes
-fno-stack-protector	No	Turns off protections
-fstack-protector	Yes	Adds to funcs that call <code>alloca()</code> & w/ arrays larger than 8 chars (<code>--param=ssp-buffer-size changes 8</code>)
-fstack-protector-strong	No	Also funcs w/ any arrays & refs to local frame addresses
-fstack-protector-all	No	All funcs

Detecting buffer overflows: stack canaries



How to circumvent canary protection?

<http://www.phrack.org/issues.html?issue=56&id=5>

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 int f (char ** argv)
6 {
7     int pipa; // useless variable
8     char *p;
9     char a[30];
10
11    p=a;
12
13    printf ("p=%x\t -- before 1st strcpy\n",p);
14    strcpy(p,argv[1]);           // <== vulnerable strcpy()
15    printf ("p=%x\t -- after 1st strcpy\n",p);
16    strncpy(p,argv[2],16);
17    printf("After second strcpy ;)\n");
18 }
19
20
21 int main(int argc, char* argv[] )
22 {
23     greeting( argv[1] );
24     printf( "Bye %s\n", argv[1] );
25 }
```

“Reading” the stack to recover canary



Request (can trigger buffer overflow in stack)

Apache forks
off child process
to handle request

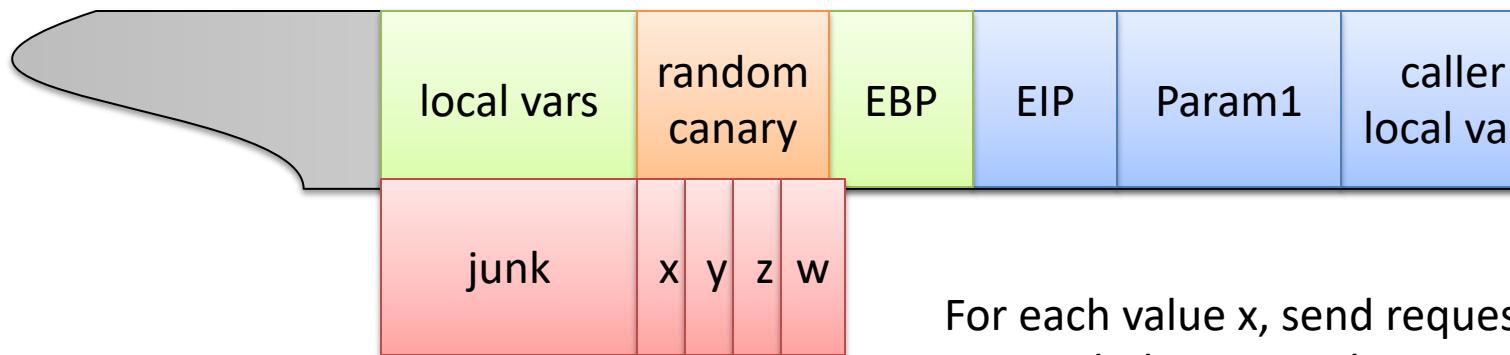


fork() copies process memory into
child. So every child has:

- same canaries
- same address randomization

Response or process crashes

Apache web server w/ buffer overflow

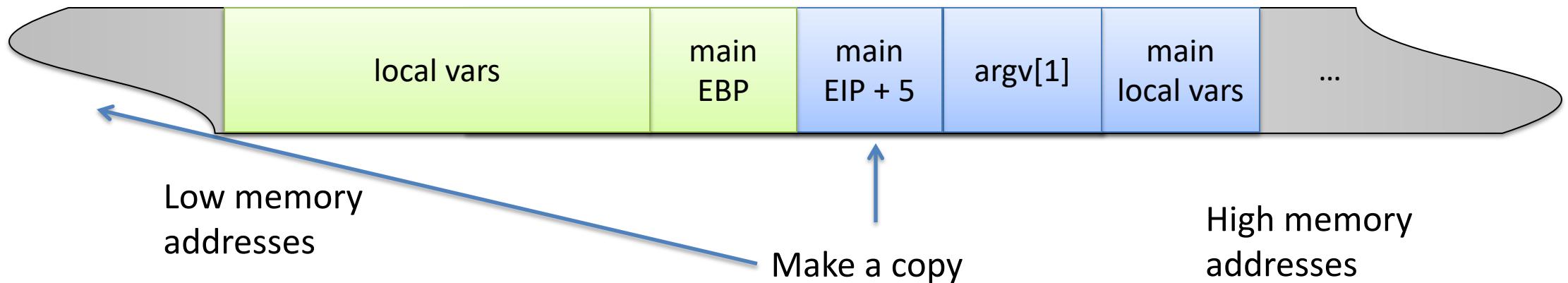


For each value x, send request and see if
responded to properly

Expected $2^7 + 2^7 + 2^7 + 2^7 = 512$ requests

Repeat for subsequent bytes of canary

Detection: copying values to safer location



StackShield:

- Function call: copy return address to safer location (beginning of .data)
- Check if stack value is different on function exit

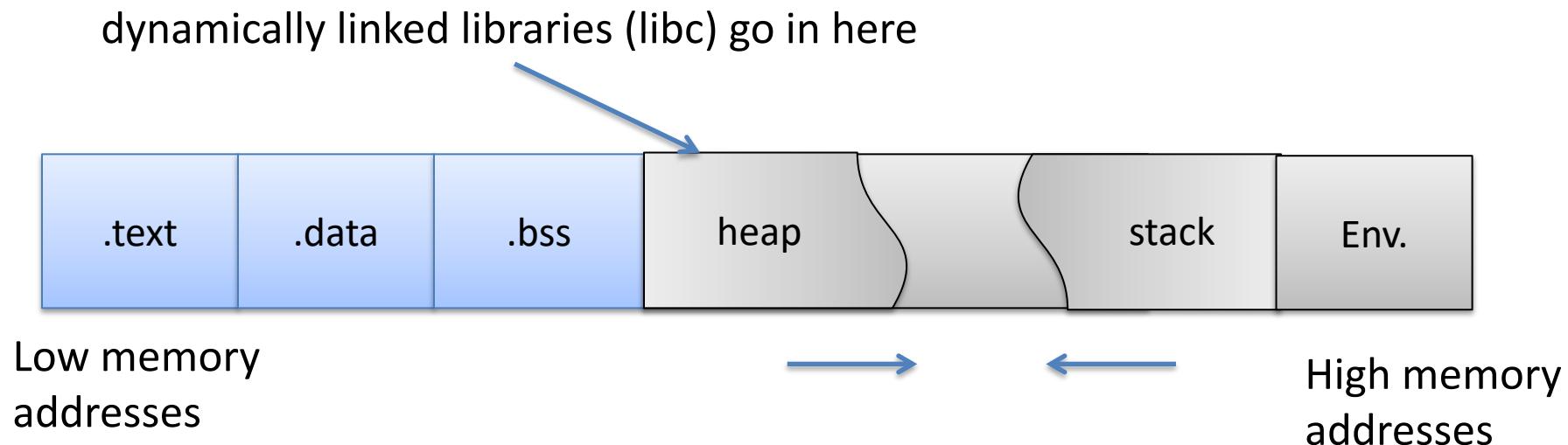
Circumvention:

- Write over saved copy & return address, if possible
- Hijack control flow without return address

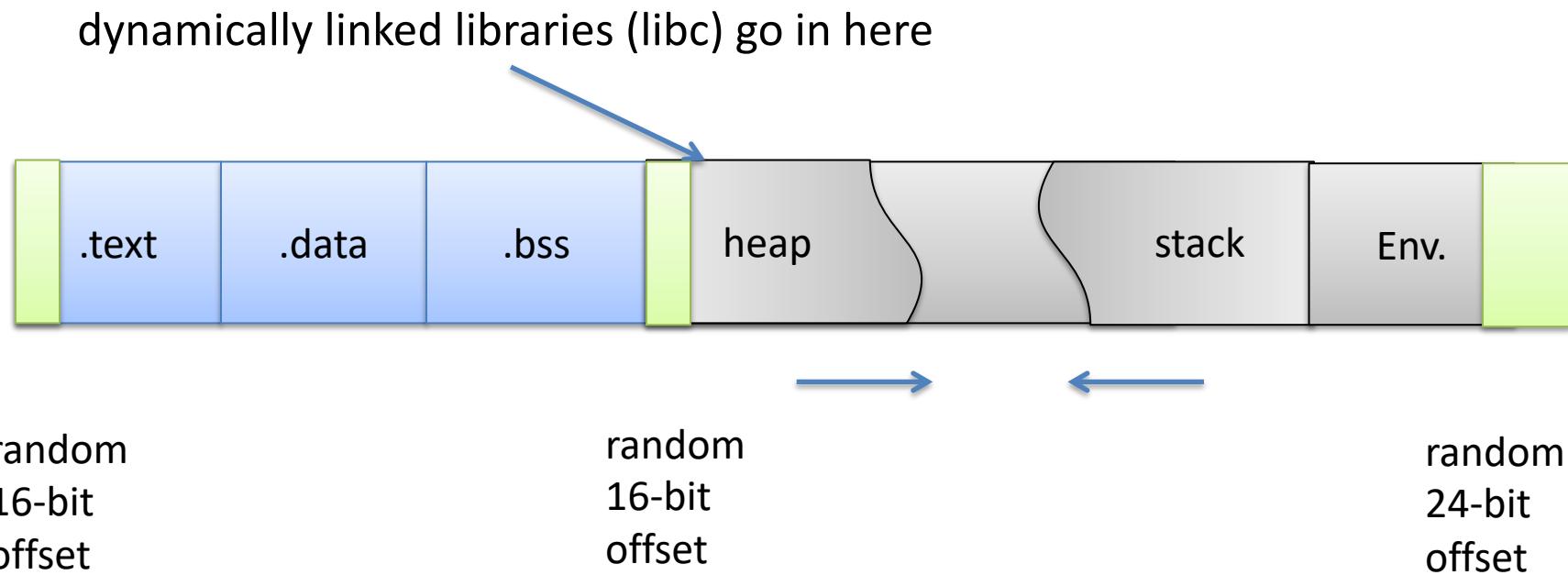
Countermeasures

- Stack protections
- Address space layout randomization
- W^X (non-executable stack)
- Control-flow integrity

Address space layout randomization (ASLR)



Address space layout randomization (ASLR)



Linux PaX implementation for example:

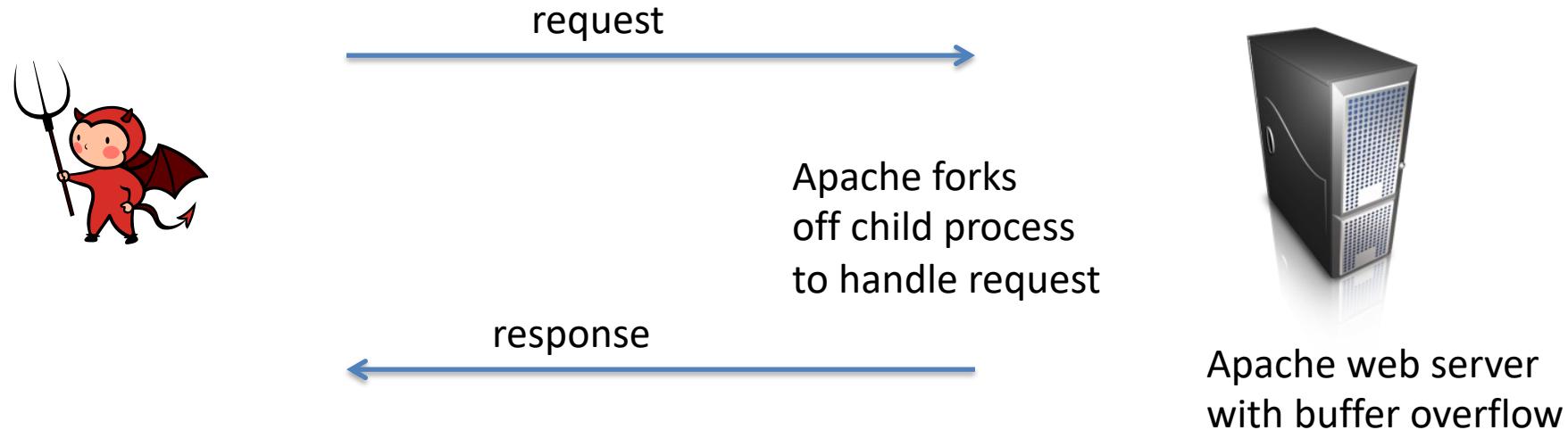
- Randomize offsets of three areas
- 16 bits, 16 bits, 24 bits of randomness. More in 64-bit architectures
- Adds some unpredictability that should help prevent exploits

Defeating ASLR

- Large nop sled with classic buffer overflow
 - W^X prevents this, stay tuned
- Use a vulnerability that can be used to leak address information
 - E.g., printf arbitrary read
- Brute force the address using forking process

Defeating ASLR

Brute-forcing example from reading “On the effectiveness of Address Space Layout Randomization” by Shacham et al.



Defeating ASLR

Brute-forcing example from reading “On the effectiveness of Address Space Layout Randomization” by Shacham et al.



Attacker makes a
guess of where usleep()
is located in memory

request

top of stack (higher addresses)
:
0x01010101
0xDEADBEEF
guessed address of usleep()
0xDEADBEEF
64 byte buffer, now filled with A's
:
bottom of stack (lower addresses)



Apache web server
with buffer overflow

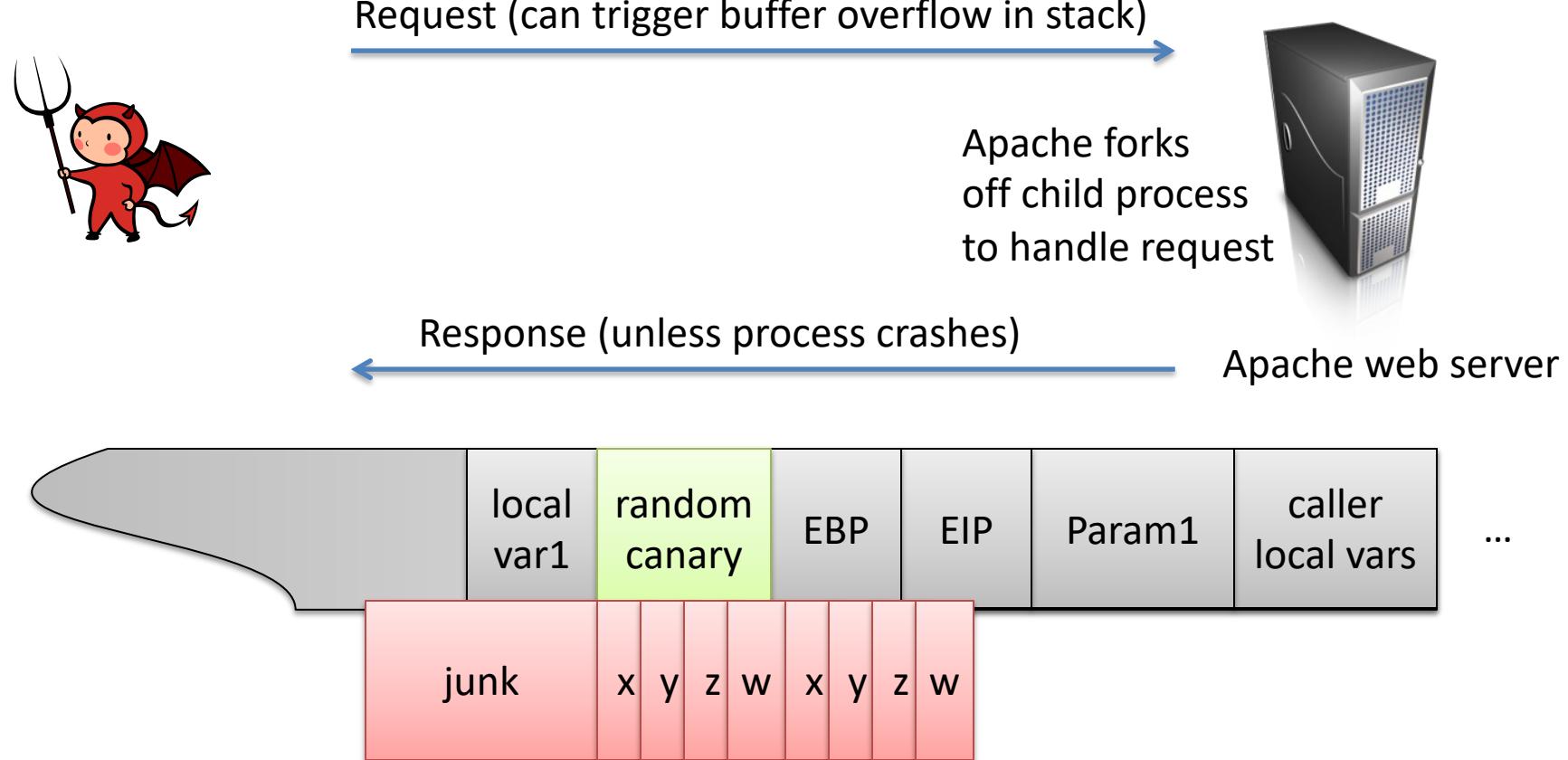
Figure 2: Stack after one probe

Failure will crash the child process
immediately and therefore kill connection

Success will crash the child process
after sleeping for 0x01010101
Microseconds and kill connection

If on 64-bit architecture, such brute-force attack unlikely to work

Reading the stack, remotely



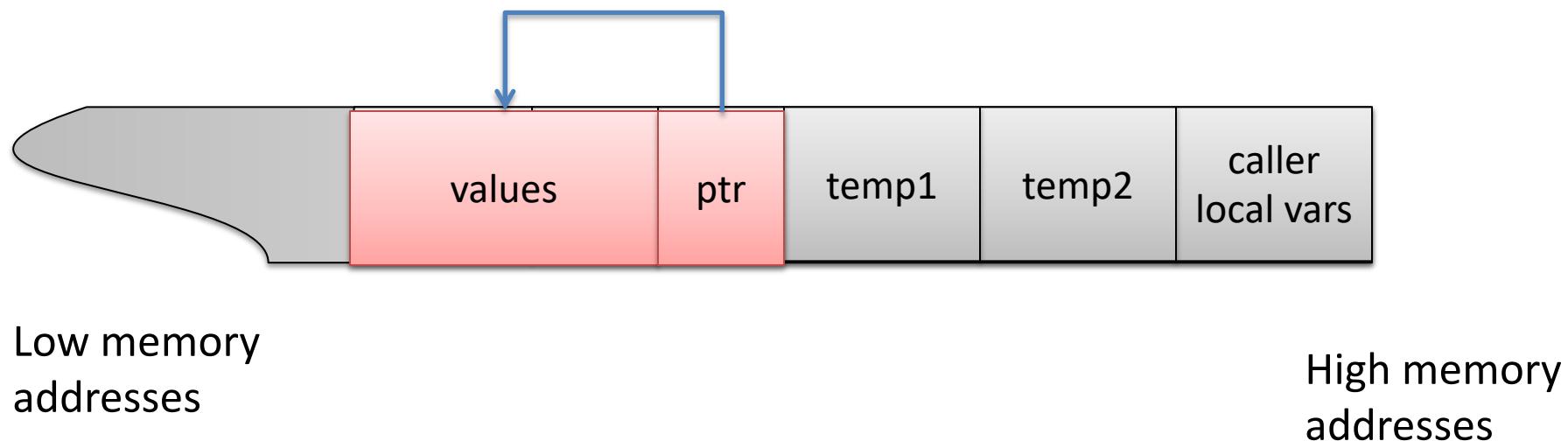
Reading stack for EBP/EIP can give approximate address offset

Countermeasures

- Stack protections
- Address space layout randomization
- W^X (non-executable stack)
- Control-flow integrity

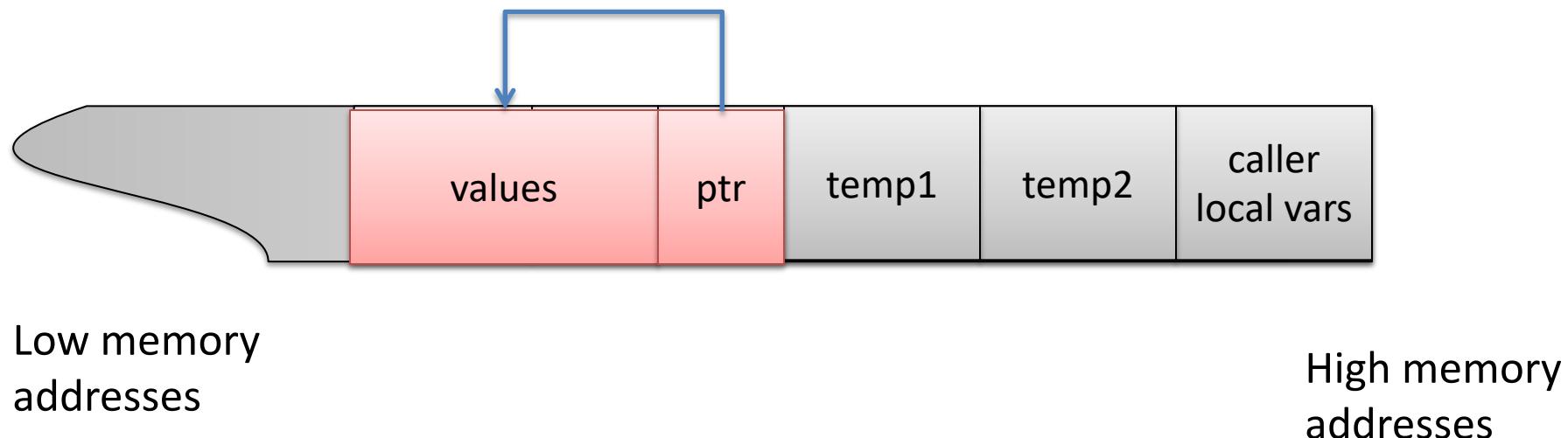
W^X (W xor X)

- The idea: mark memory page as either
 - Writable or Executable (not both)
- Specifically: make heap and stack non-executable
- Default in gcc now, turn off with `-z execstack`



W^X (W xor X)

- AMD64: NX bit (Non-Executable)
IA-64: XD bit (eXecute Disabled)
ARMv6: XN bit (eXecute Never)
 - Extra bit in each page table entry
 - Processor refuses to execute code if bit = 1
 - Mark heap and stack segments as such



W^X (W xor X)

Software emulation of NX bits

- ExecShield (RedHat Linux)
- PaX (Page-eXec) (uses NX bit if available)

mprotect()

- Process can set permissions on memory pages

Will W^X prevent:

- AlephOne's stack overflow exploit? Yes
- Stack smash that overwrites pointer to point at shell code in Heap or Env variable? Yes
- Heap overflow with same shell location? Yes
- Double free with same shell location? Yes

Return-into-libc exploits

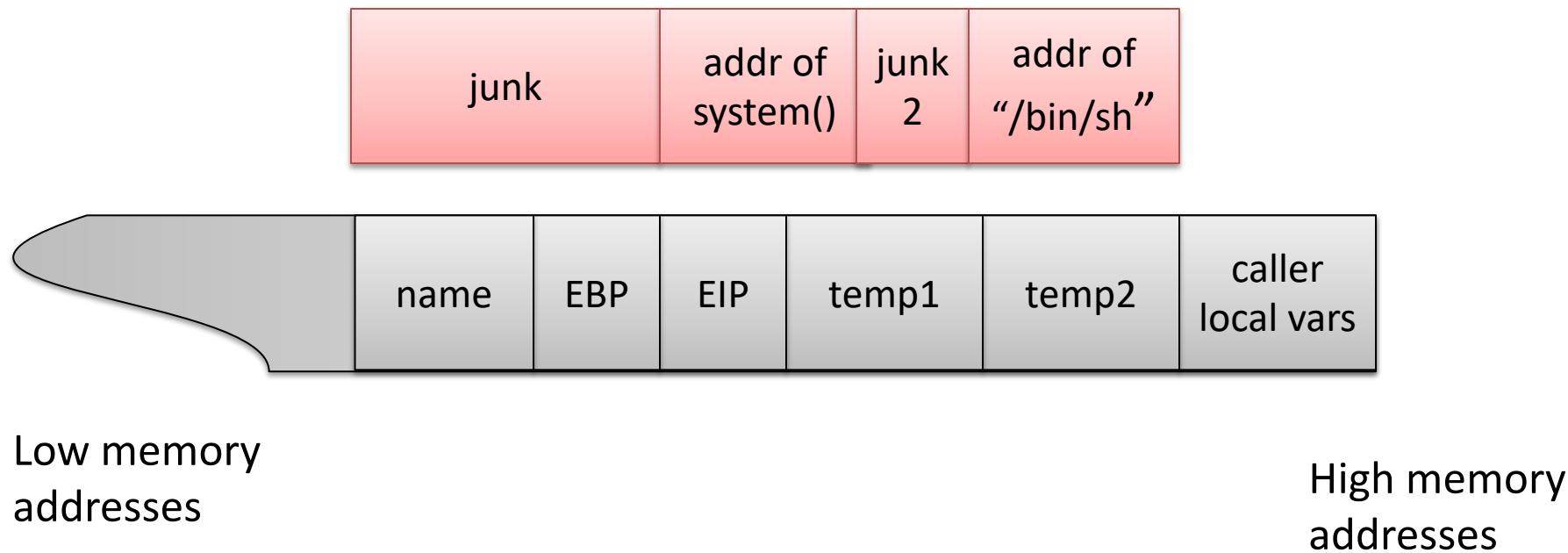
- libc is standard C library, included in all processes
- system() --- execute commands on system

```
(gdb) b main
Breakpoint 1 at 0x80484a0: file sploit1.c, line 15.
(gdb) r
Starting program: /home/user/pp1/sploits/sploit1

Breakpoint 1, main () at sploit1.c:15
15      args[0] = TARGET;
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ecf180 <system>
(gdb) _
```

Return-into-libc exploits

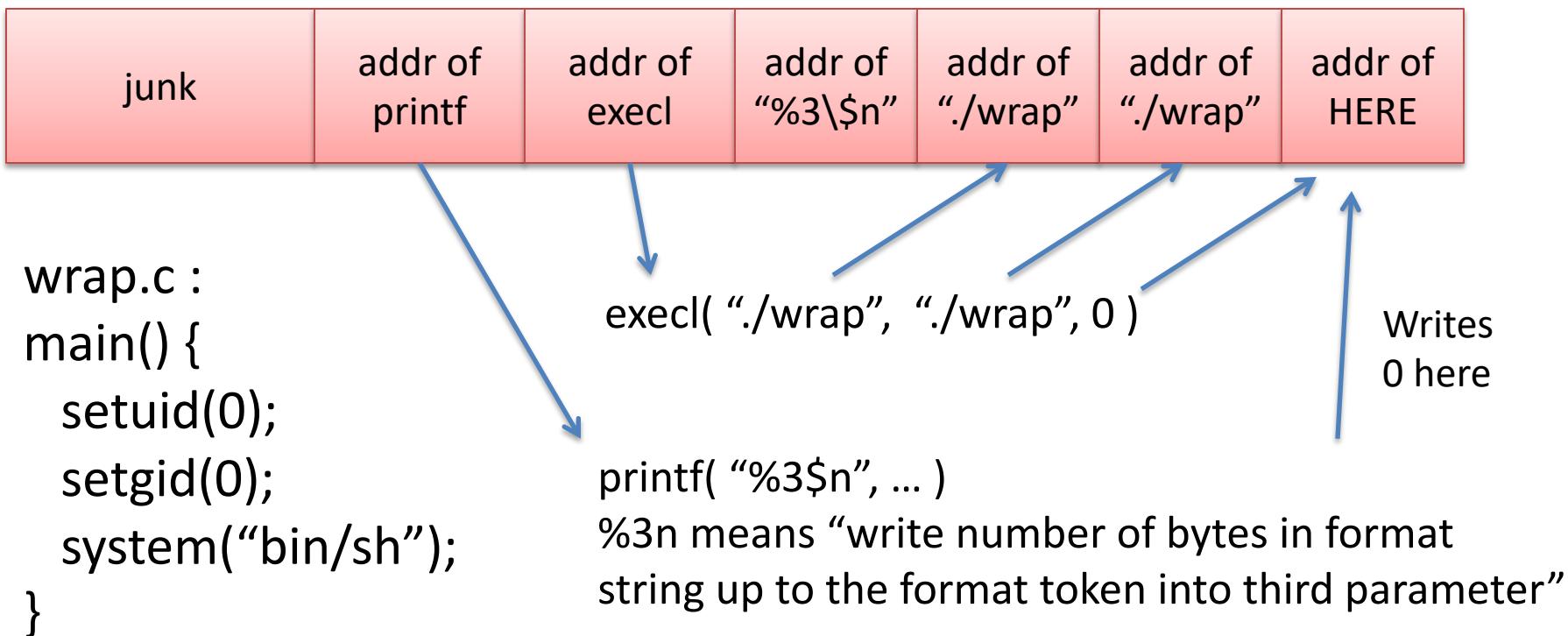
Overwrite EIP with address of system() function
junk2 just some filler: returned to after system call
first argument to system() is ptr to “/bin/sh”



Return-into-libc exploits

This simple exploit has a few deficiencies (for attacker):

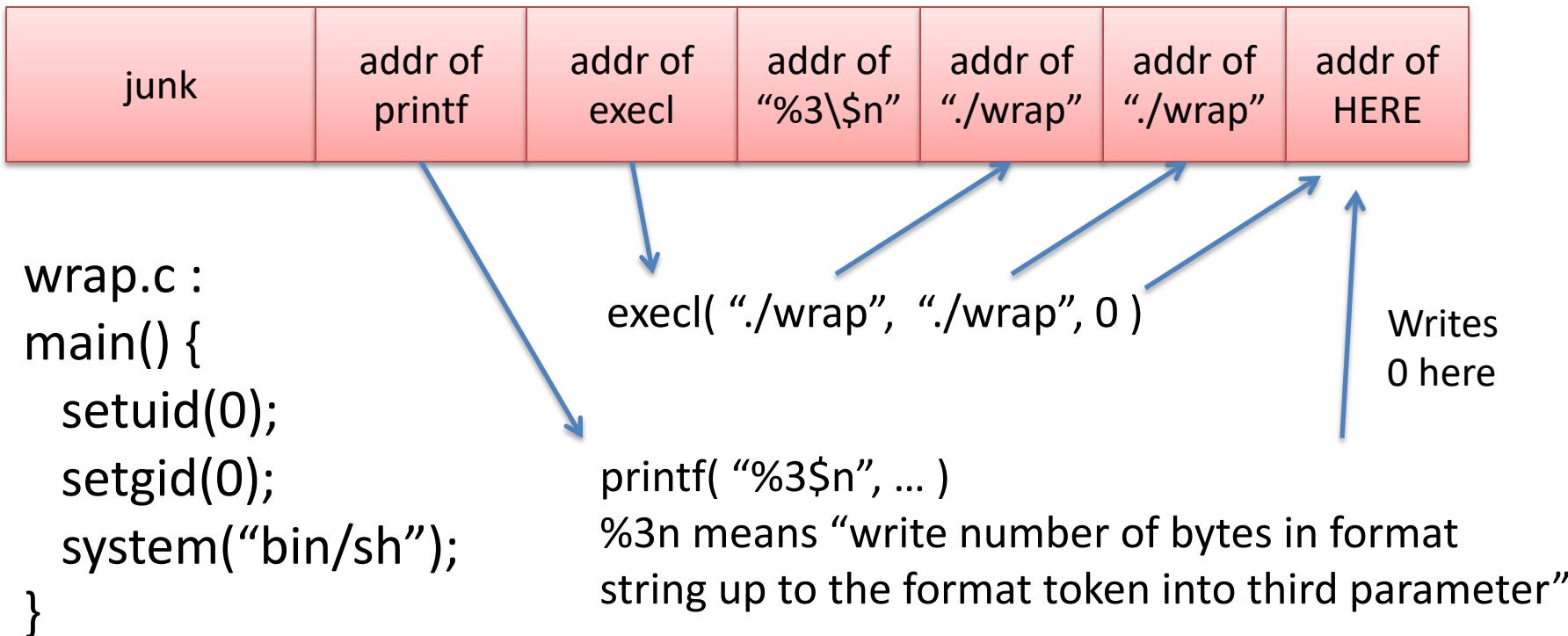
- Crashes after exiting called /bin/sh (easy to fix with exit())
- system() drops privileges by default



Return-into-libc exploits

These exploits only execute instructions marked executable

W^X cannot stop such an attack



Return-into-libc exploits

Return-into-libc may seem limited:

- Only useful for calling libc functions
- Okay in last example, but not always sufficient
- Before W^X, exploit could run arbitrary code

Can we ***not*** inject any malicious code and yet have an exploit that runs ***arbitrary*** code?

Return-oriented programming (ROP)

- Second return-into-libc exploit:
 - self-modifying exploit buffer to call a sequence of libc calls
- ROP chains together sequences of calls to “gadgets”
 - Gadget = sequence of instructions that do something useful
- Can find gadgets in executable code pages (e.g., stdlib)
- Shacham showed that stdlib is Turing-complete

Countermeasures

- Stack protections
- Address space layout randomization
- W^X (non-executable stack)
- Control-flow integrity

Control-flow integrity

- Abadi et al. 2005
- Recall: control flow graph is (static) set of paths in program
- Include checks that call is found in control-flow graph
 - Simplest: memory locations callable or not (one bit)
 - More complex: label memory locations with ID, use to match-up caller-callee edges in CFG
- Versions implemented in Microsoft Visual Studio, clang

Countermeasures

- Stack protections
- Address space layout randomization
- W^X (non-executable stack)
- Control-flow integrity

Arms race between exploit designers and countermeasures

- Much harder to exploit some vulnerabilities
- Costs more to develop exploits

Next time: finding vulnerabilities in software

Return-oriented programming (ROP)

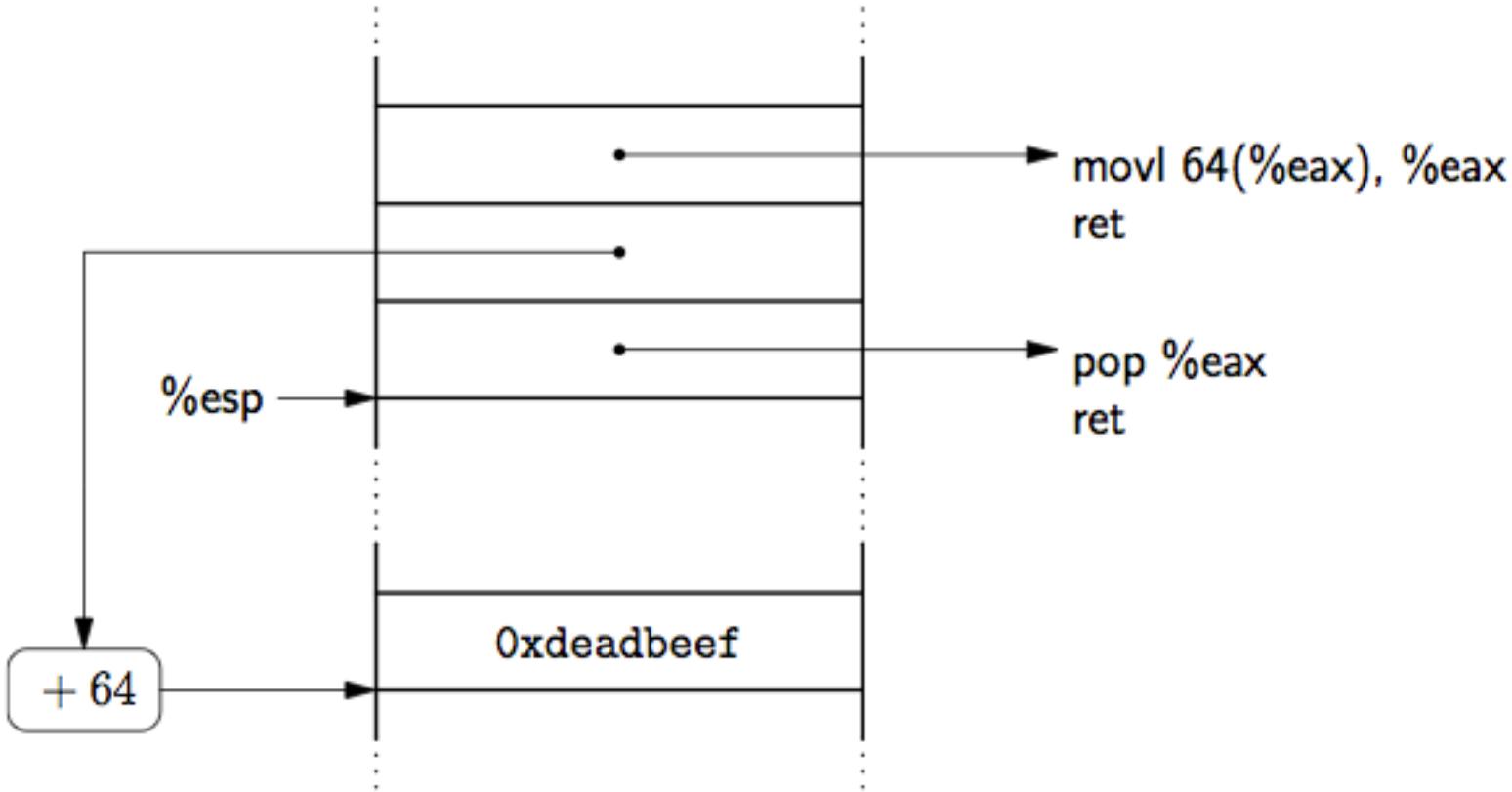
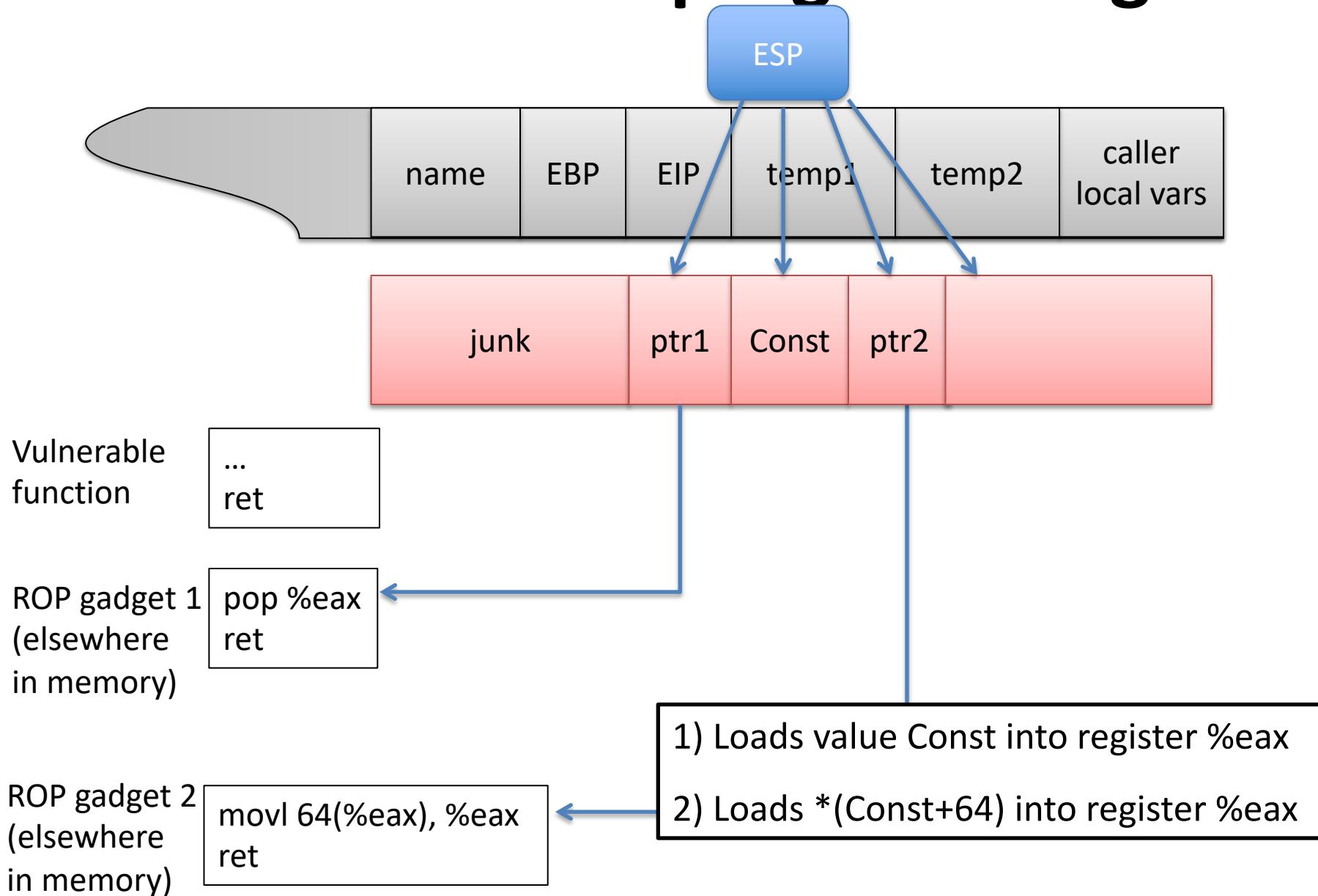


Figure 3: Load a word in memory into `%eax`.

From Shacham “The Geometry of Innocent Flesh on the Bone...” 2007

Return-oriented programming



Return-oriented programming (ROP)

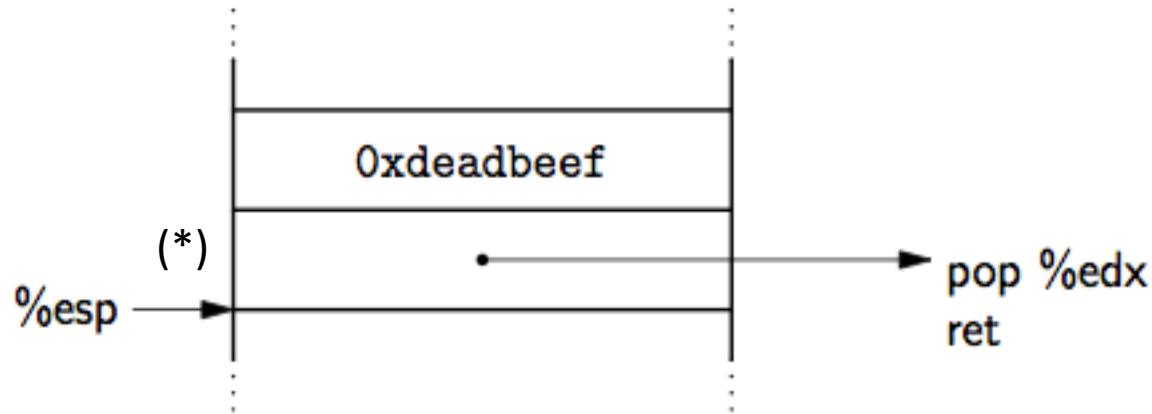


Figure 2: Load the constant **0xdeadbeef** into **%edx**.

From Shacham “The Geometry of Innocent Flesh on the Bone...” 2007

If this is on stack and (*) is return pointer after buffer overflow, then the result will be loading **0xdeadbeef** into **edx** register

From
Shacham
“The Geometry of
Innocent Flesh on
the Bone...” 2007

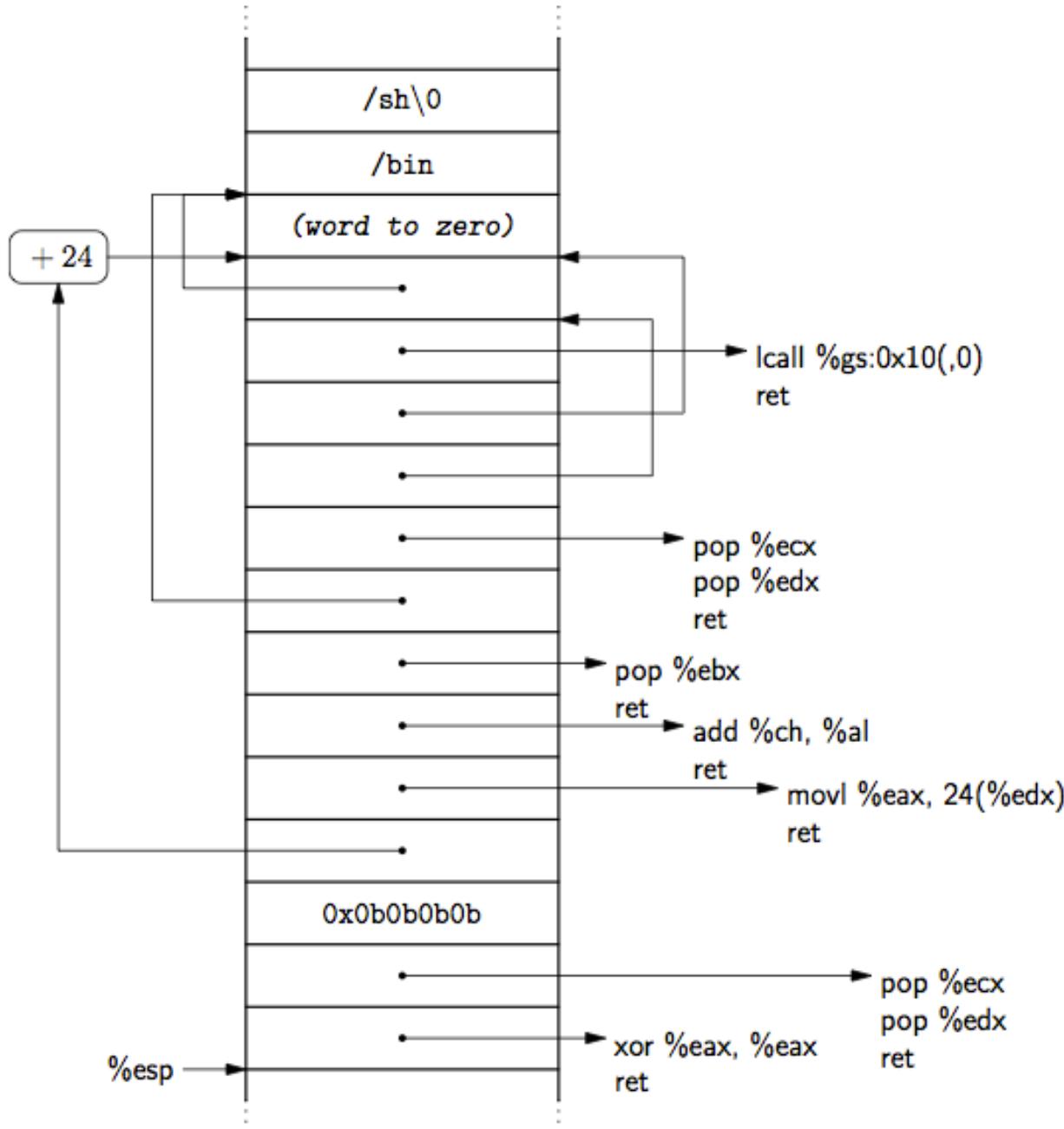


Figure 16: Shellcode.

ROP where do we get code snippets?

movl \$0x00000001, -44(%ebp)	c7 45 d4 01 00 00 00	}	add %dh, %bh
test \$0x00000007, %edi	f7 c7 07 00 00 00 00	}	movl \$0x0F000000, (%edi)
setnz -61(%ebp)	0f 95 45 c3	}	xchg %ebp, %eax inc%ebp ret

