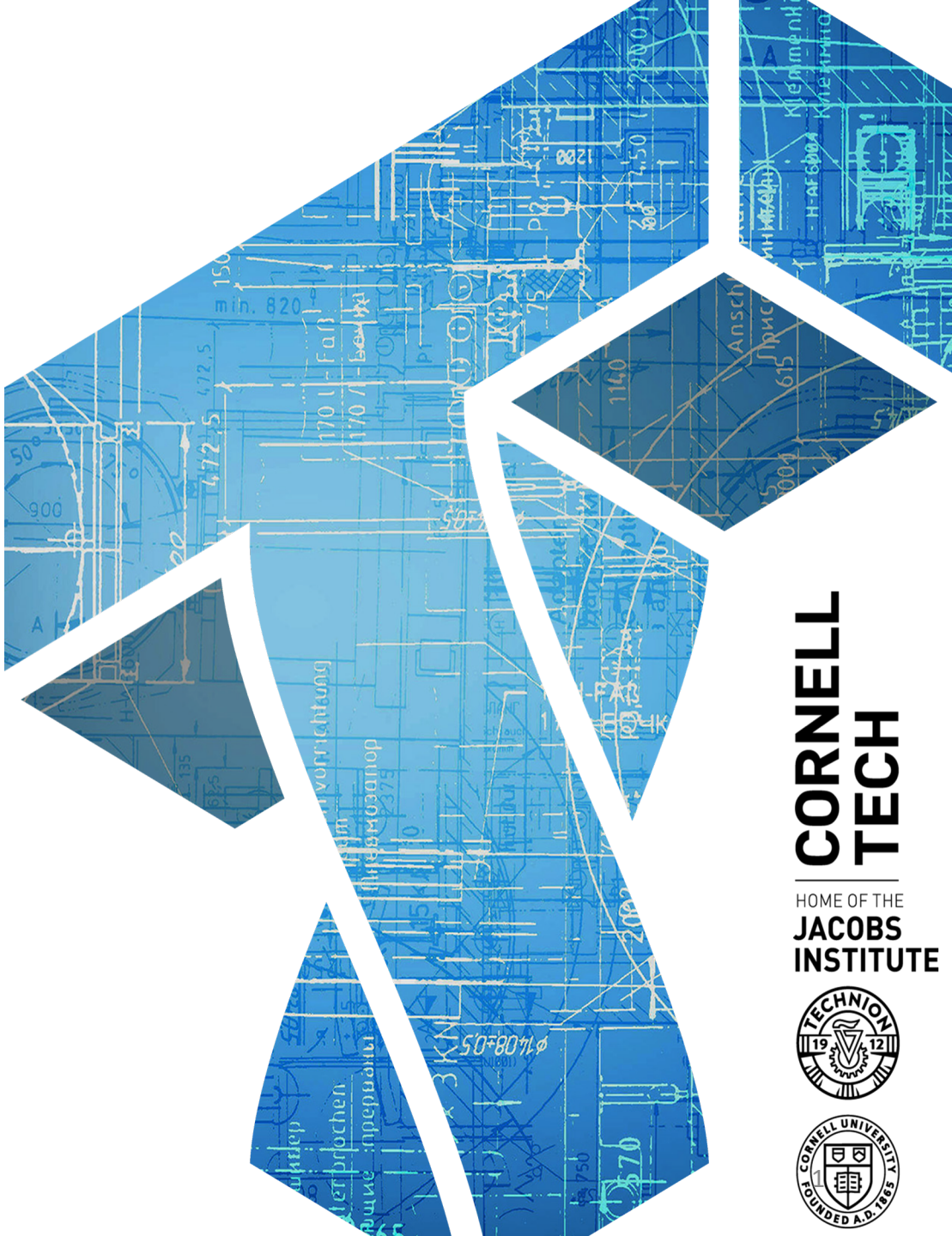
CS 5435: Memory protection mechanisms

Instructor: Tom Ristenpart

https://github.com/tomrist/cs5435-fall2019



CORNELL TECH

HOME OF THE JACOBS INSTITUTE

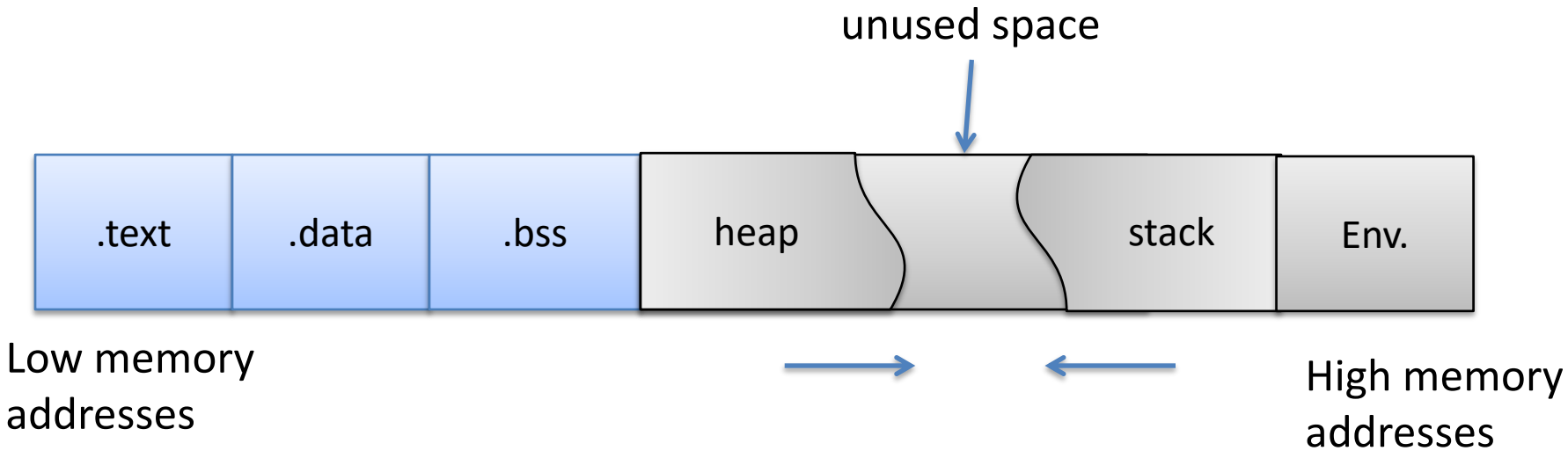
Running demo example
(modified from Gray hat hacking book linked in resources)

```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] );
}
```

Process memory layout



.text:
machine code of executable

.data:
global initialized variables

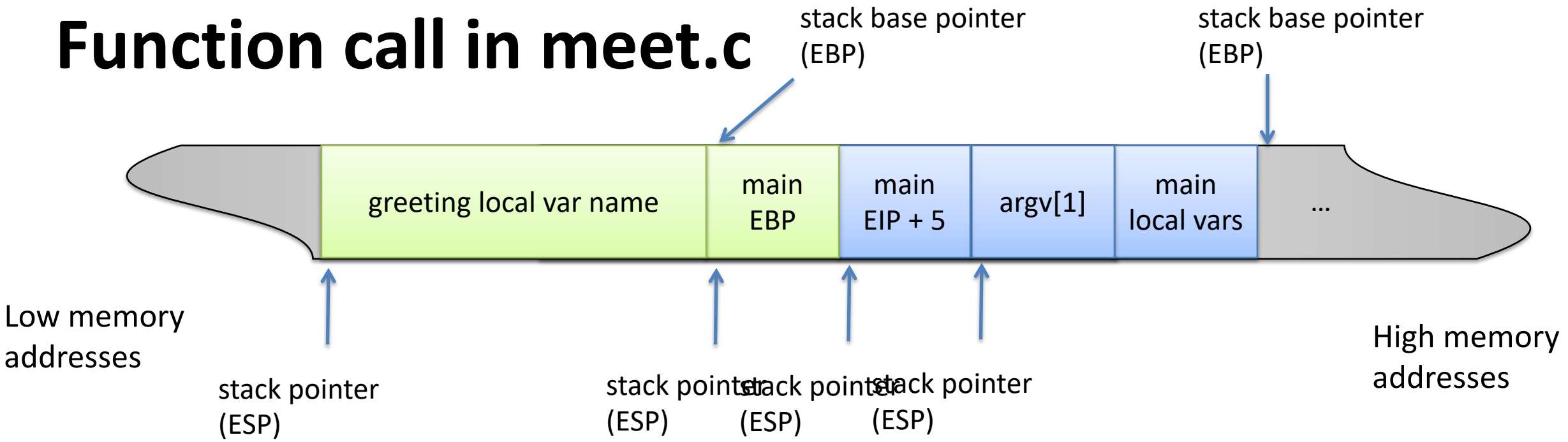
.bss:
"below stack section"
global uninitialized variables

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

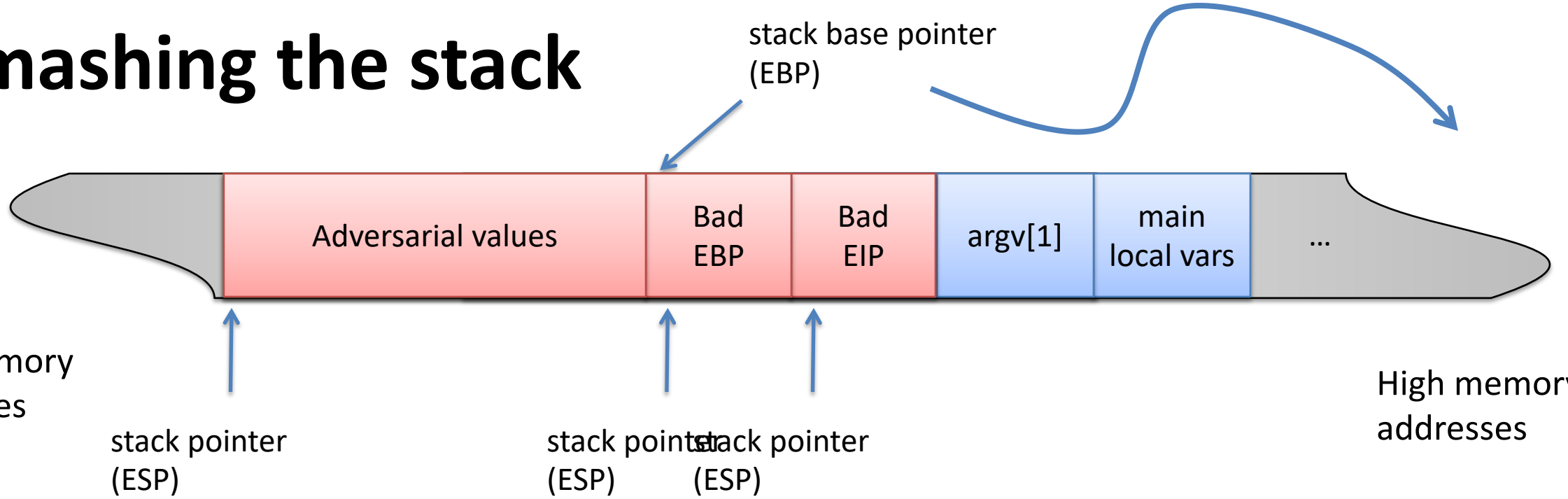
…. (more stuff including strcpy) …

```
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
```

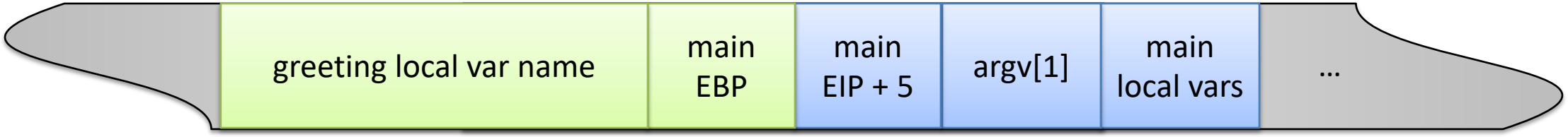
.... (more stuff including strcpy) ...

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

eip

eip

Countermeasures?

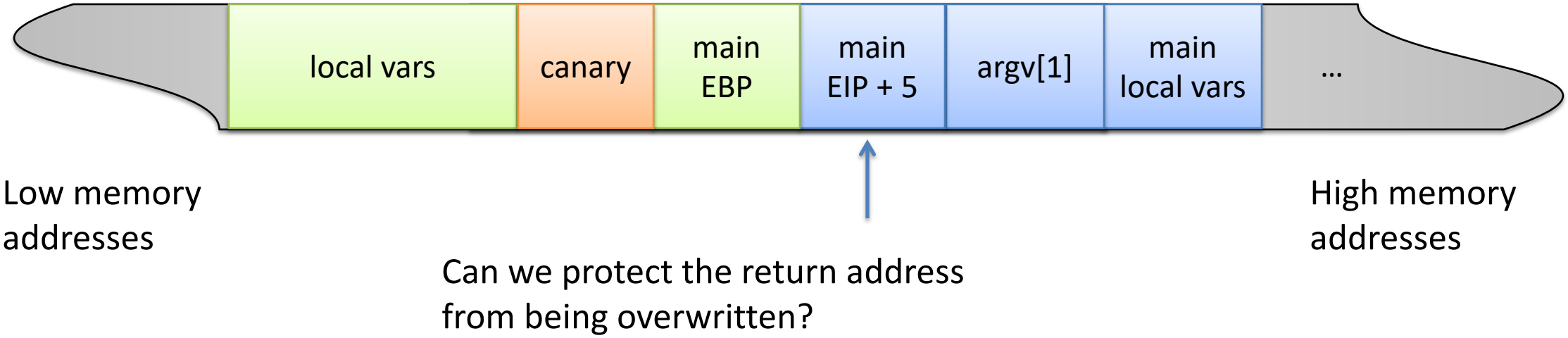


Low memory addresses

High memory addresses

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

Detecting buffer overflows: stack canaries



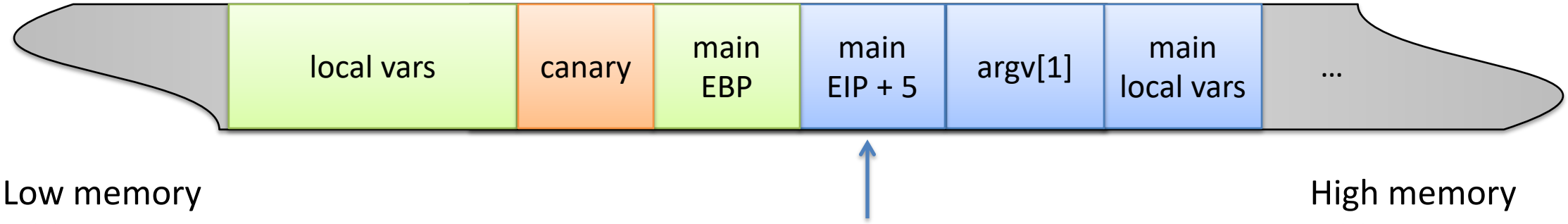
Before executing function, write canary value to stack

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):

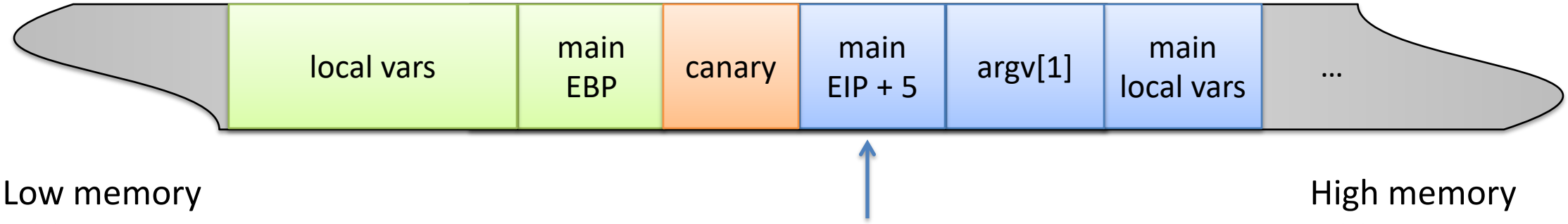
- GCC extension that adds runtime canary checking between saved EIP and EBP
- 8% overhead on Apache

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):

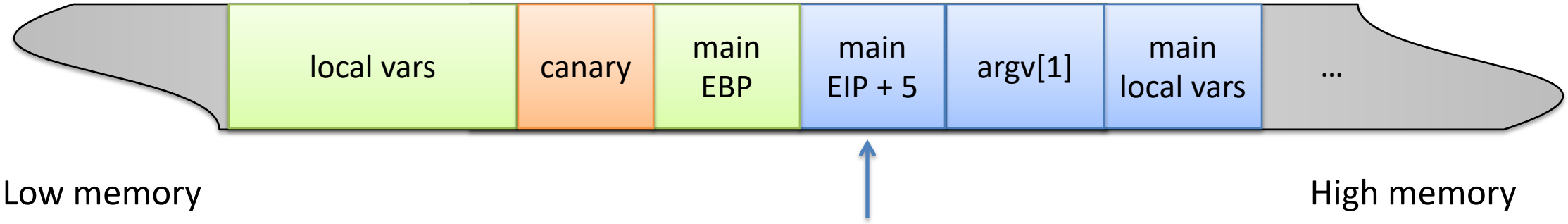
- GCC extension that adds runtime canary checking between saved EIP and EBP
- 8% overhead on Apache

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

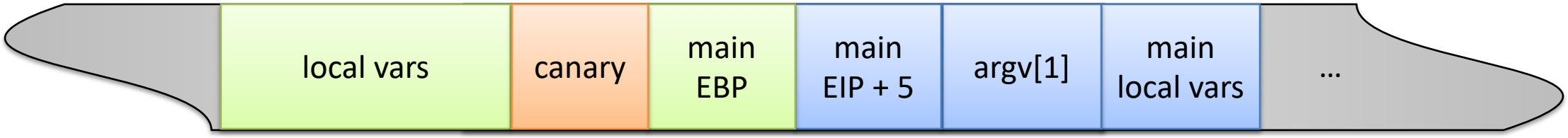
What is the problem with this?

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



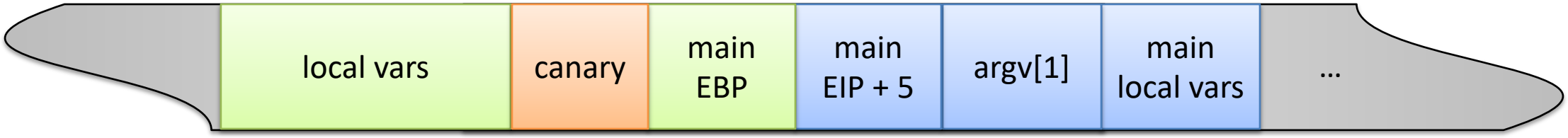
Low memory addresses

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

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


int f (char ** argv)
{
  int pipa; // useless variable
  char *p;
  char a[30];

  p=a;

  printf ("p=%x\t -- before 1st strcpy\n",p);
  strcpy(p,argv[1]);          // <== vulnerable strcpy()
  printf ("p=%x\t -- after 1st  strcpy\n",p);
  strncpy(p,argv[2],16);
  printf("After second strcpy ;)\n");
}


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

“Reading” the stack to recover canary

Request (can trigger buffer overflow in stack)

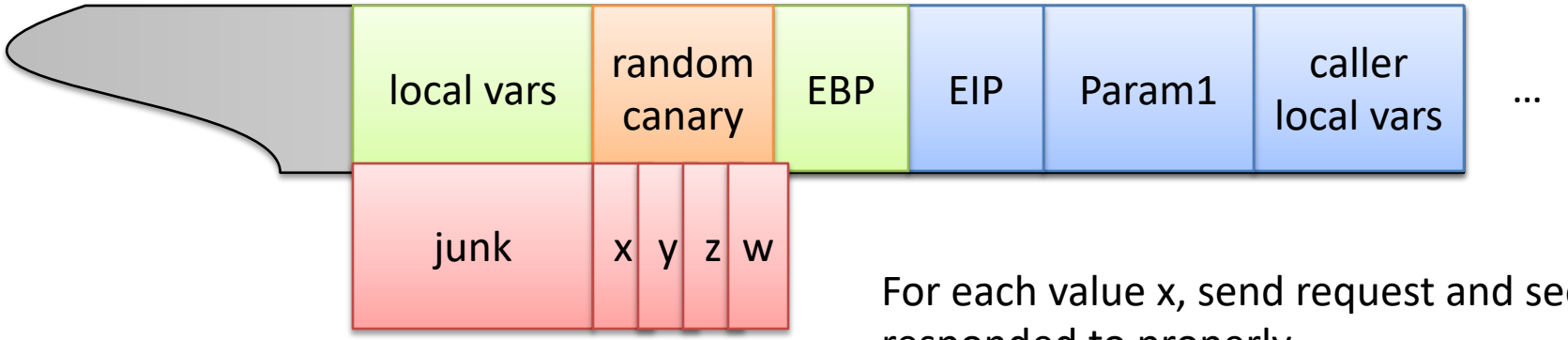
Apache forks
off child process
to handle request

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

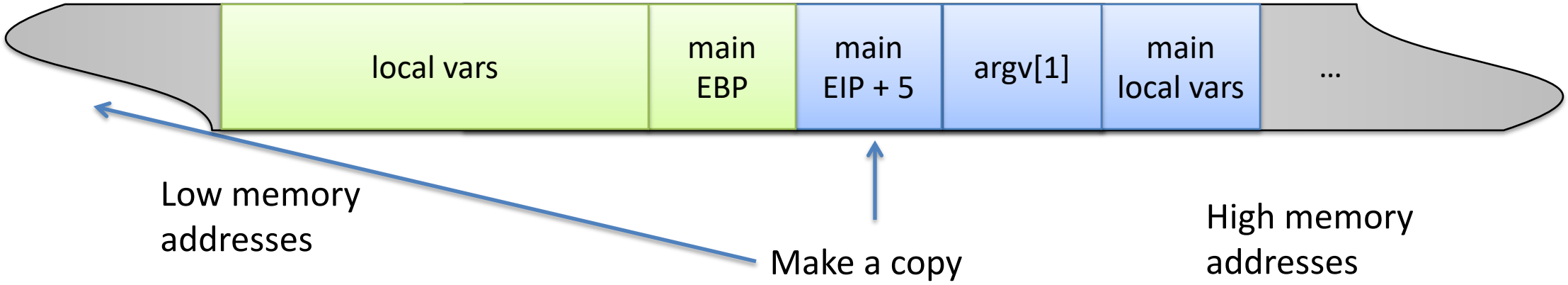


For each value x, send request and see if responded to properly

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

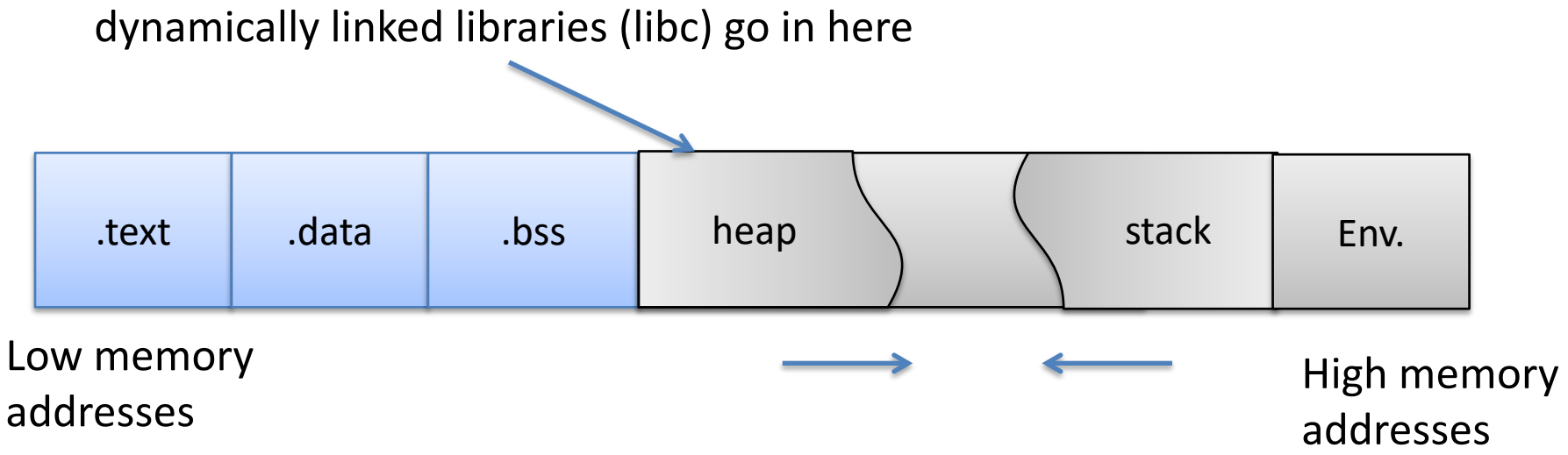
Circumvention:

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

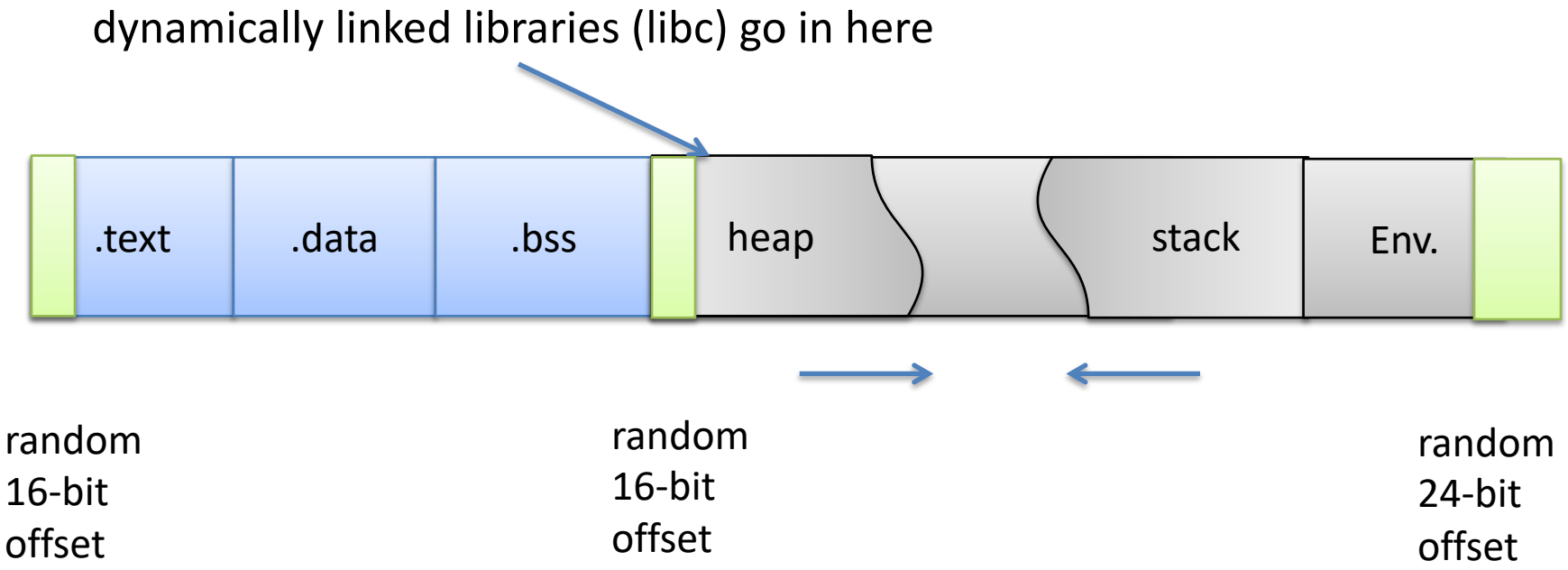
Countermeasures

- 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.

request

Apache forks
off child process
to handle request

response

Apache web server
with buffer overflow

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

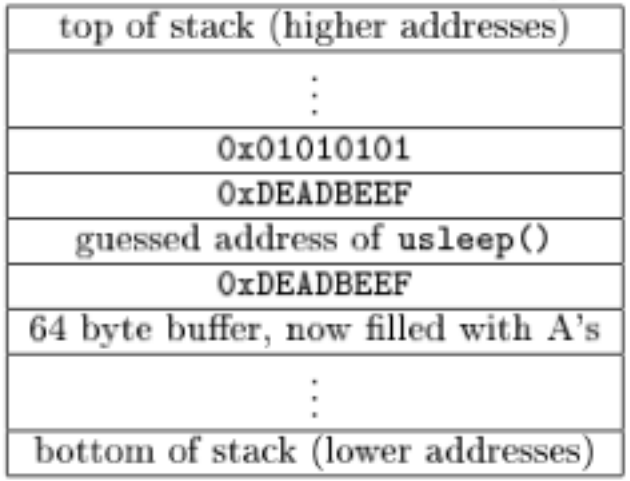


Figure 2: Stack after one probe

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

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

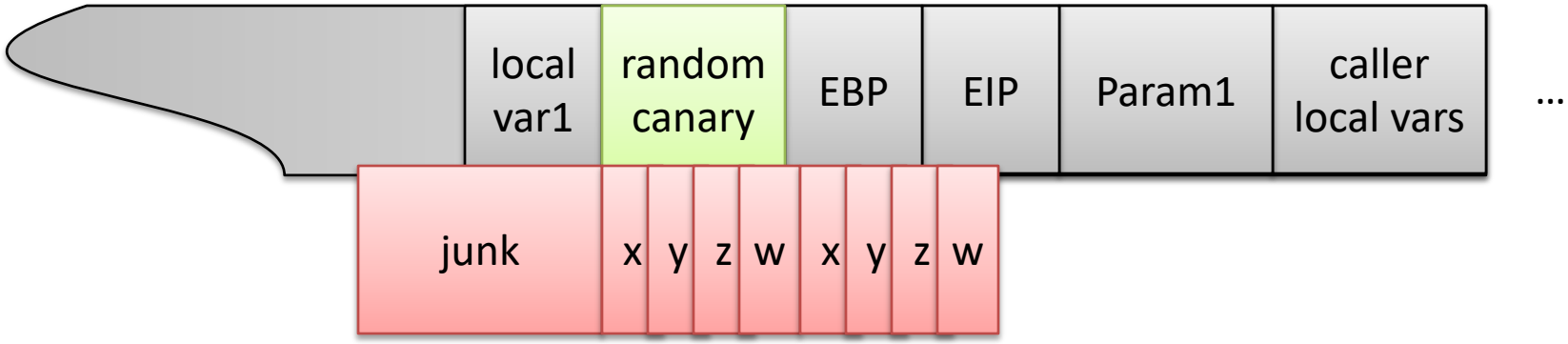
Reading the stack, remotely

Request (can trigger buffer overflow in stack)

Apache forks
off child process
to handle request

Response (unless process crashes)

Apache web server



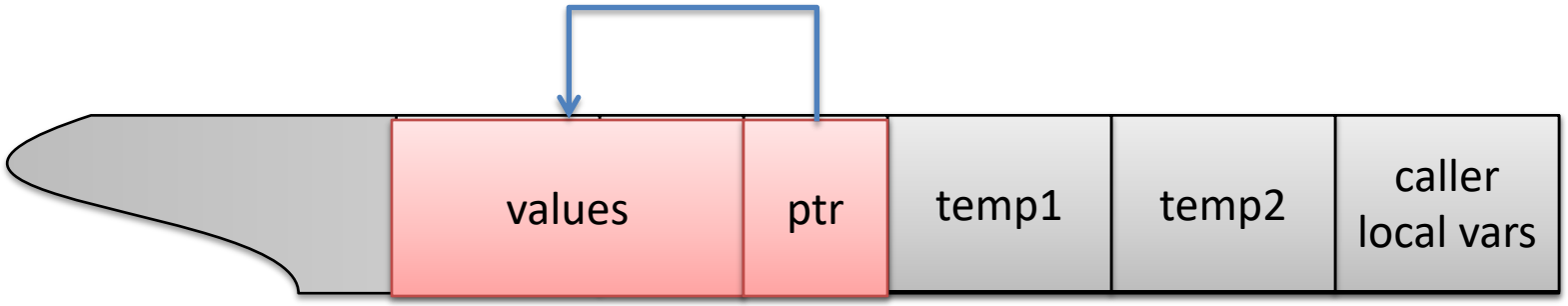
Reading stack for EBP/EIP can give approximate address offset

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)

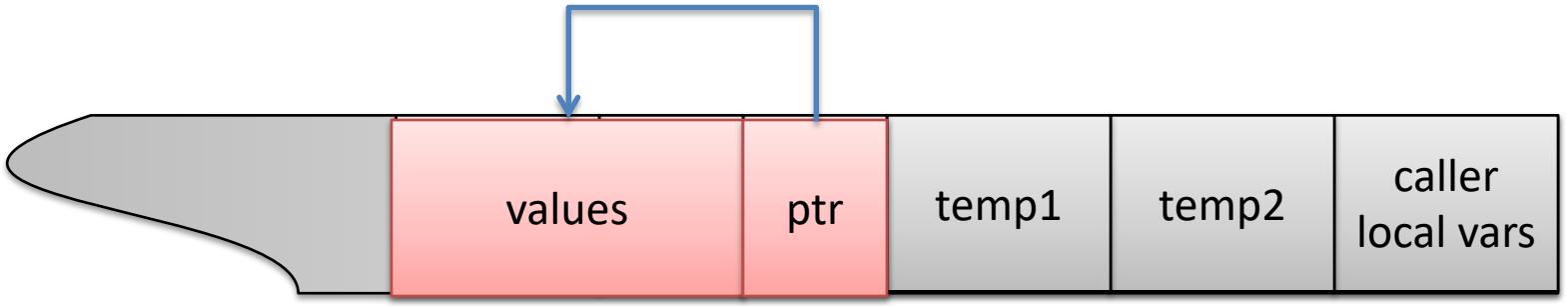
  IA-64: XD bit (eXecute Disabled)

  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

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

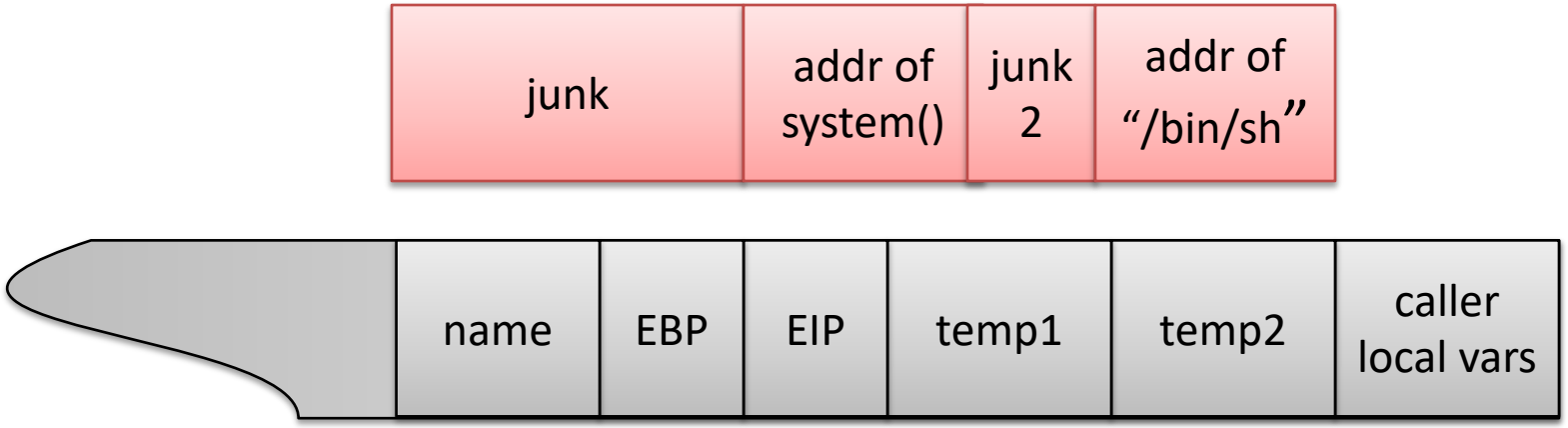
- libc is standard C library, included in all processes
- system() --- execute commands on system

```
(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

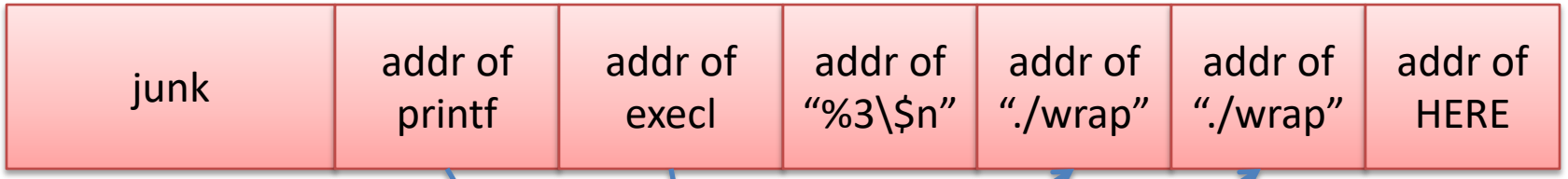
Overwrite EIP with address of system() function
junk2 just some filler: returned to after system call
first argument to system() is ptr to “/bin/sh”



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




```
wrap.c :
main() {
  setuid(0);
  setgid(0);
  system("bin/sh");
}
```

execl("./wrap", "./wrap", 0)

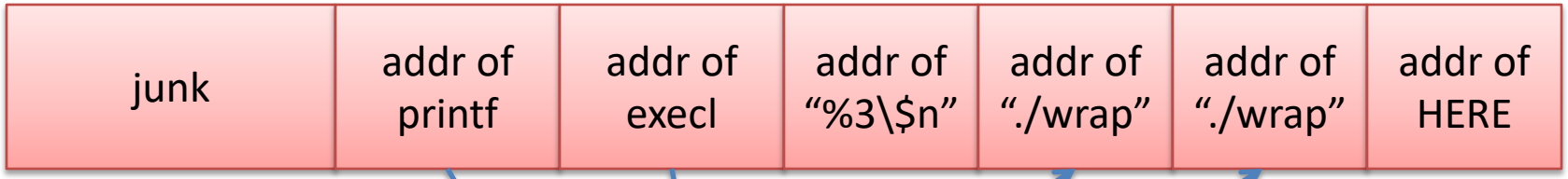
Writes 0 here

printf("%3$n", …)
%3n means "write number of bytes in format string up to the format token into third parameter"

Return-into-libc exploits

These exploits only execute instructions marked executable

W^X cannot stop such an attack




```
wrap.c :
main() {
  setuid(0);
  setgid(0);
  system("bin/sh");
}
```

execl("./wrap", "./wrap", 0)

Writes 0 here

printf("%3$n", …)
%3n means "write number of bytes in format string up to the format token into third parameter"

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)

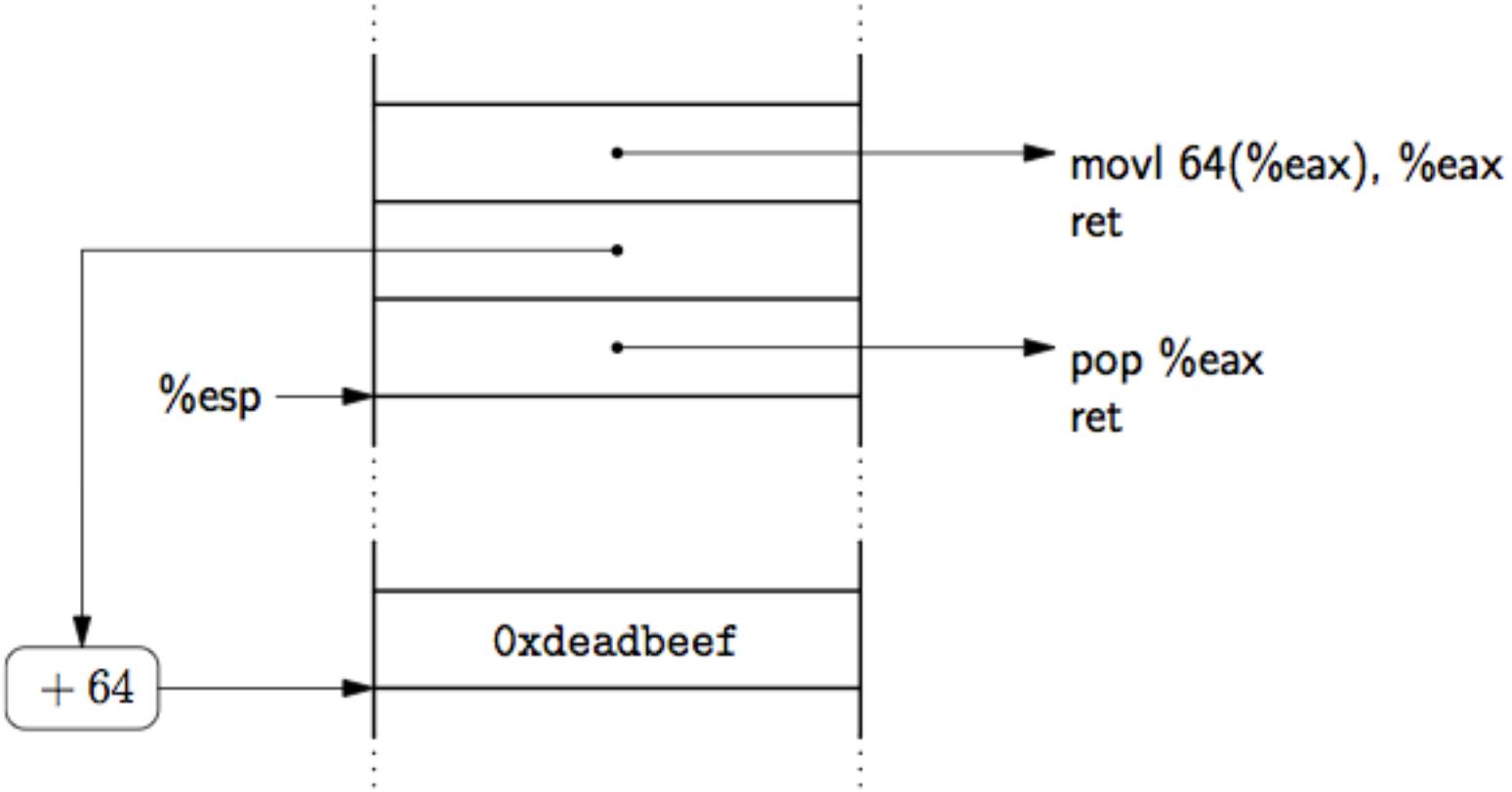
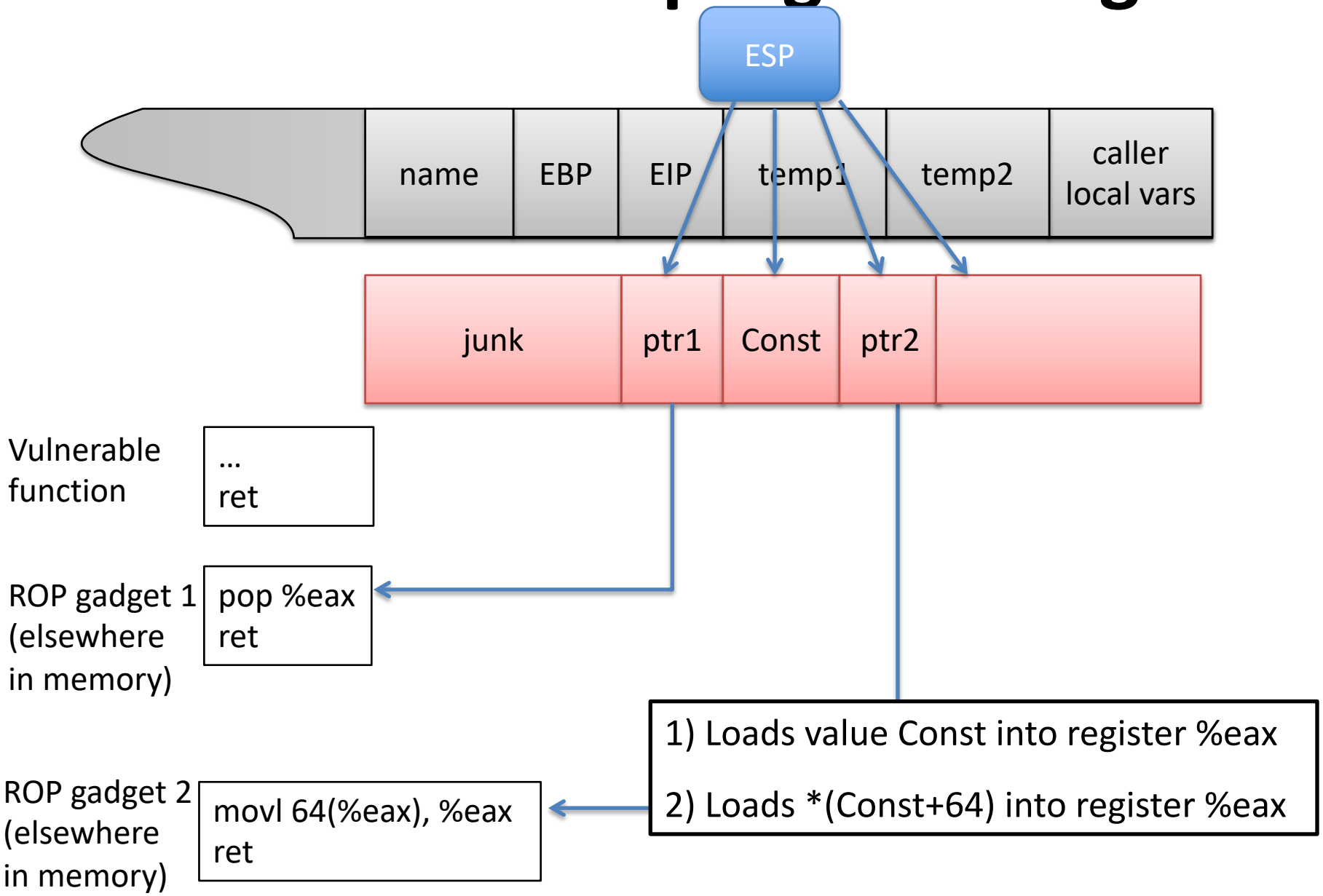


Figure 3: Load a word in memory into %eax.

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

Return-oriented programming



Return-oriented programming (ROP)

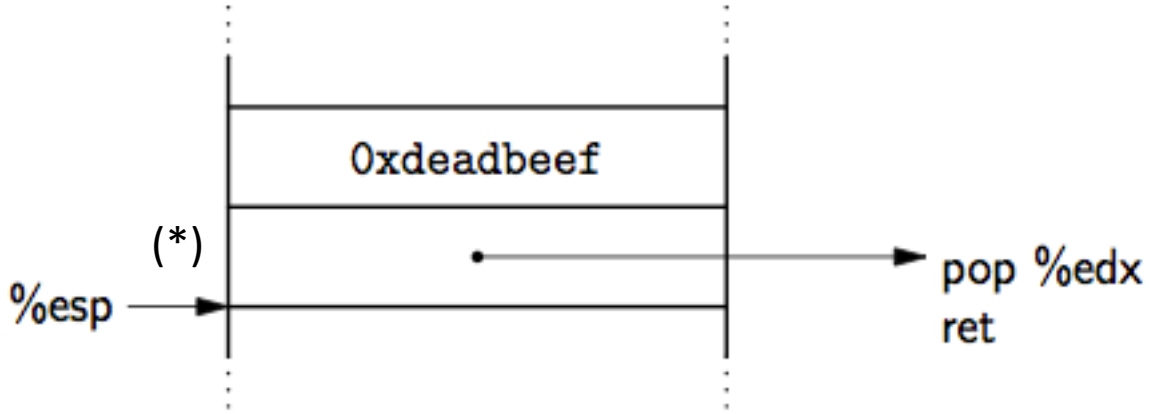


Figure 2: Load the constant 0xdeadbeef into %edx.

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

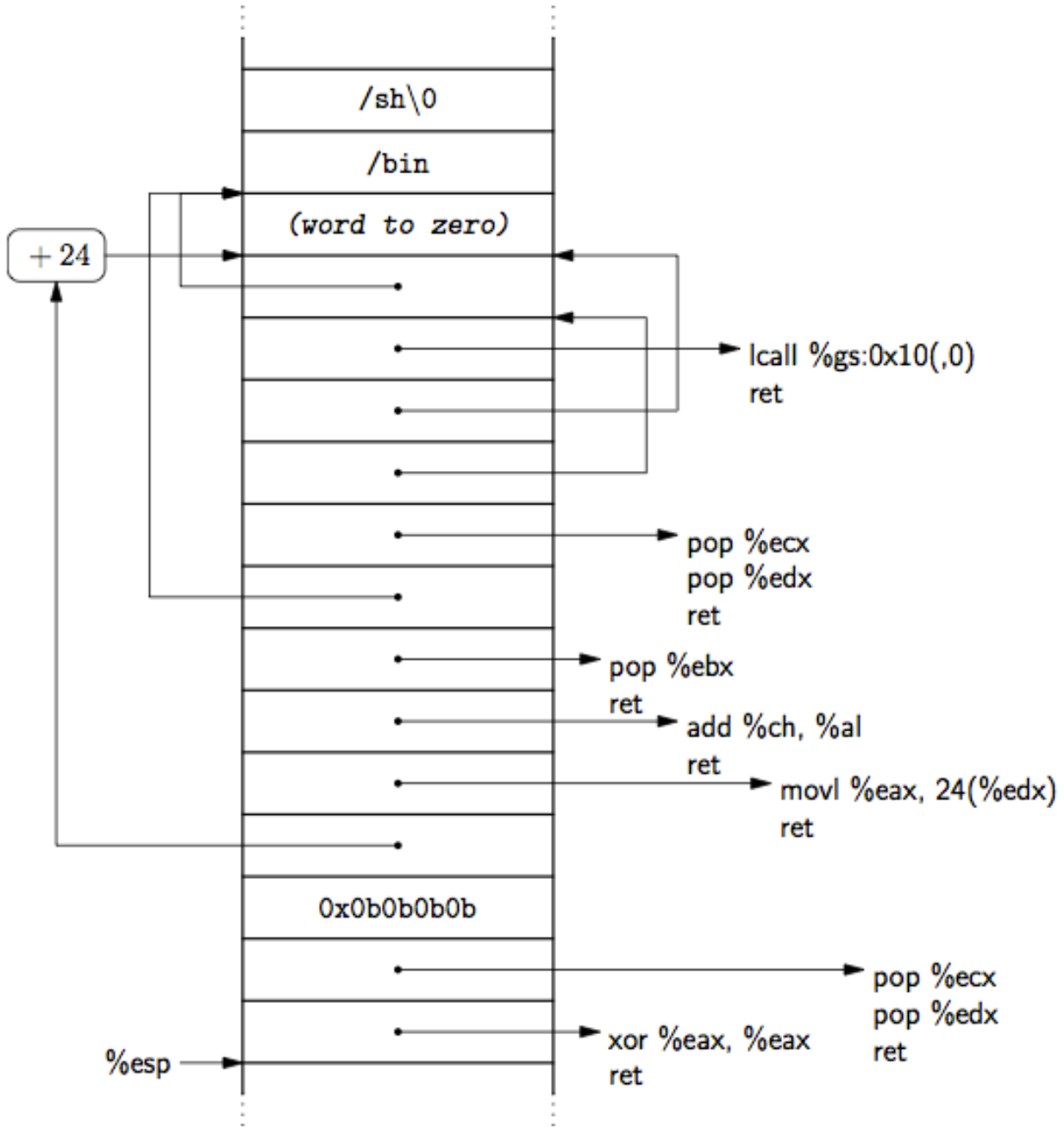
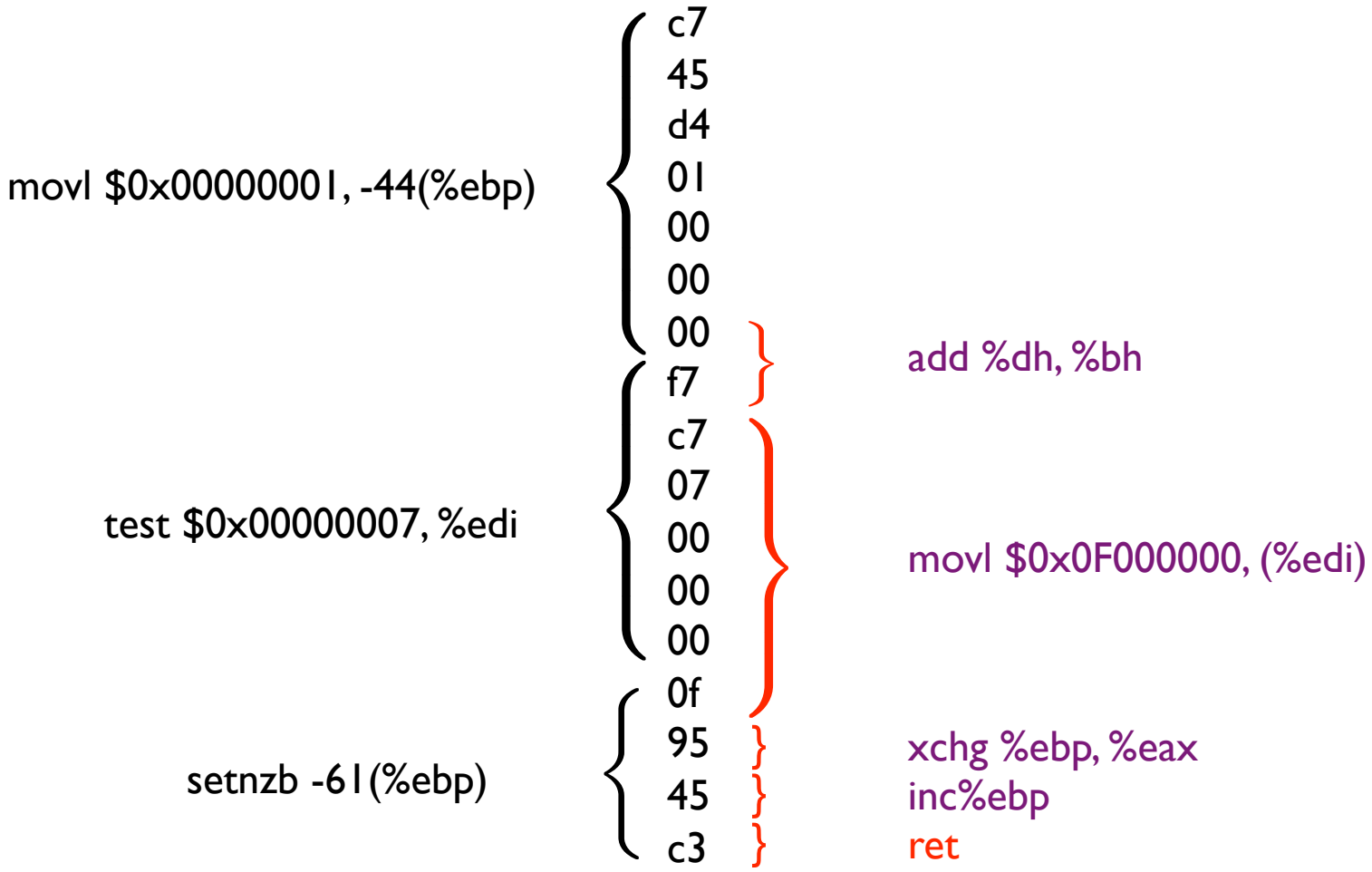


Figure 16: Shellcode.

ROP where do we get code snippets?



Buchanan et al., Blackhat 2008