

Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh,
David Mazières, Dan Boneh

Stanford University

Hacking 101

Exploit



GET /0xDEAD HTTP/1.0



shell

```
$ cat /etc/passwd  
root:x:0:0:::/bin/sh  
sorbo:x:6:9:pac:/bin/sh
```



Crash or no Crash? Enough to build exploit



GET /blabla HTTP/1.0

HTTP/1.0 404 Not Found

GET /AAAAAAAAAAAAAAAAAAAAA

connection closed



Don't even need to know what application is running!

Exploit scenarios:

1. Open source



2. Open binary



3. Closed-binary (and source)



Attack effectiveness

- Works on 64-bit Linux with ASLR, NX and canaries

Server	Requests	Time (mins)
nginx	2,401	1
MySQL	3,851	20
Toy proprietary service (unknown binary and source)	1,950	5

Attack requirements

1. Stack vulnerability, and knowledge of how to trigger it.
2. Server process that respawns after crash
 - E.g., nginx, MySQL, Apache, OpenSSH, Samba.

Background Outline

- Stack Vulnerabilities
- No-eXecute memory (NX)
- Return-Oriented Programming (ROP)
- Address Space Layout Randomization (ASLR)

Stack vulnerabilities

```
void process_packet(int s) {  
    char buf[1024];  
    int len;  
  
    read(s, &len, sizeof(len));  
    read(s, buf, len);  
  
    return;  
}
```

handle_client()

Stack:

return address
0x400000

buf[1024]



Stack vulnerabilities

```
void process_packet(int s) {  
    char buf[1024];  
    int len;  
  
    read(s, &len, sizeof(len));  
    read(s, buf, len);  
  
    return;  
}
```

handle_client()

Stack:

return address
0x400000

AAAAAAAAAA
AAAAAAAAAA
AAAAAAAAAA
AAAAAAAAAA

A diagram illustrating a stack overflow. On the right, a vertical stack of memory is shown. The top cell contains the text 'return address' and the hexadecimal value '0x400000'. Below it are four cells, each containing the string 'AAAAAAAAAA'. A blue arrow points from the top of the 'AAAAAAAAAA' section back to the 'return address' cell, indicating that the overflowed data has overwritten the return address. The entire stack structure is enclosed in a light blue box with a black border.

Stack vulnerabilities

```
void process_packet(int s) {  
    char buf[1024];  
    int len;  
  
    read(s, &len, sizeof(len));  
    read(s, buf, len);  
  
    return;  
}
```

Stack:

return address
0x41414141

AAAAAAAAAA
AAAAAAAAAA
AAAAAAAAAA
AAAAAAAAAA

??



Stack vulnerabilities

```
void process_packet(int s) {  
    char buf[1024];  
    int len;  
  
    read(s, &len, sizeof(len));  
    read(s, buf, len);  
  
    return;  
}
```

Stack:

return address
0x500000

Shellcode:

```
dup2(sock, 0);  
dup2(sock, 1);  
execve("/bin/sh", 0, 0);
```

AAAAAAAAAA
AAAAAAAAAA
AAAAAAAAAA
AAAAAAAAAA



Stack vulnerabilities

```
void process_packet(int s) {  
    char buf[1024];  
    int len;  
  
    read(s, &len, sizeof(len));  
    read(s, buf, len);  
  
    return;  
}
```

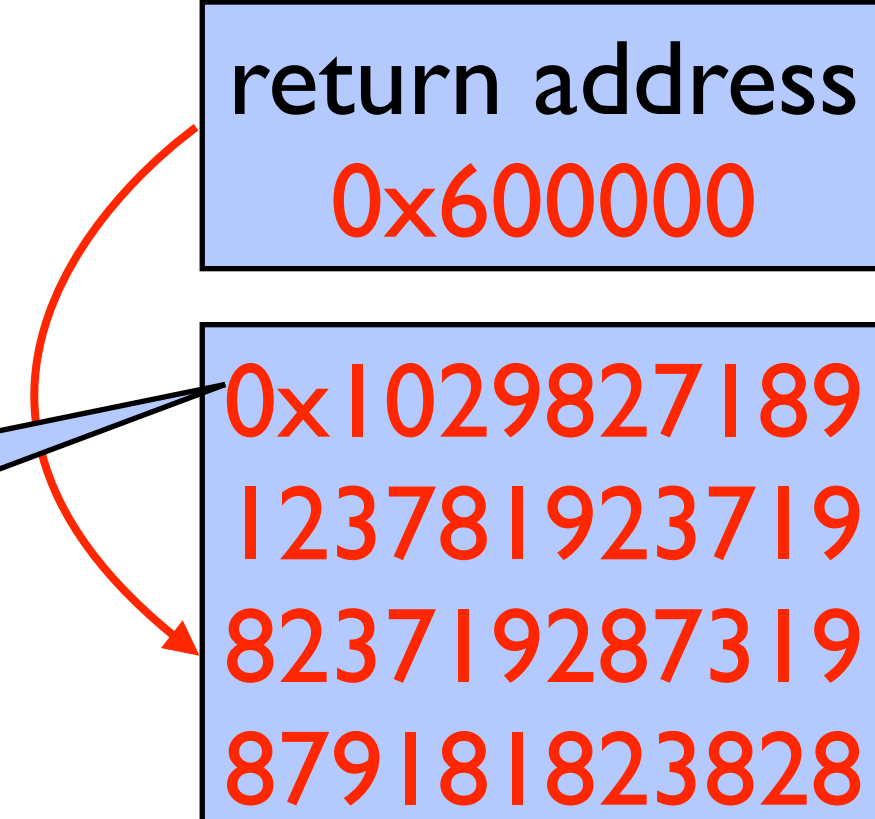
Stack:

return address
0x600000

Shellcode:

```
dup2(sock, 0);  
dup2(sock, 1);  
execve("/bin/sh", 0, 0);
```

0x1029827189
123781923719
823719287319
879181823828



Exploit protections

```
void pro  
char  
int le
```

1. Make stack non-executable (NX)

```
re  
re  
return,
```

2. Randomize memory addresses (ASLR)

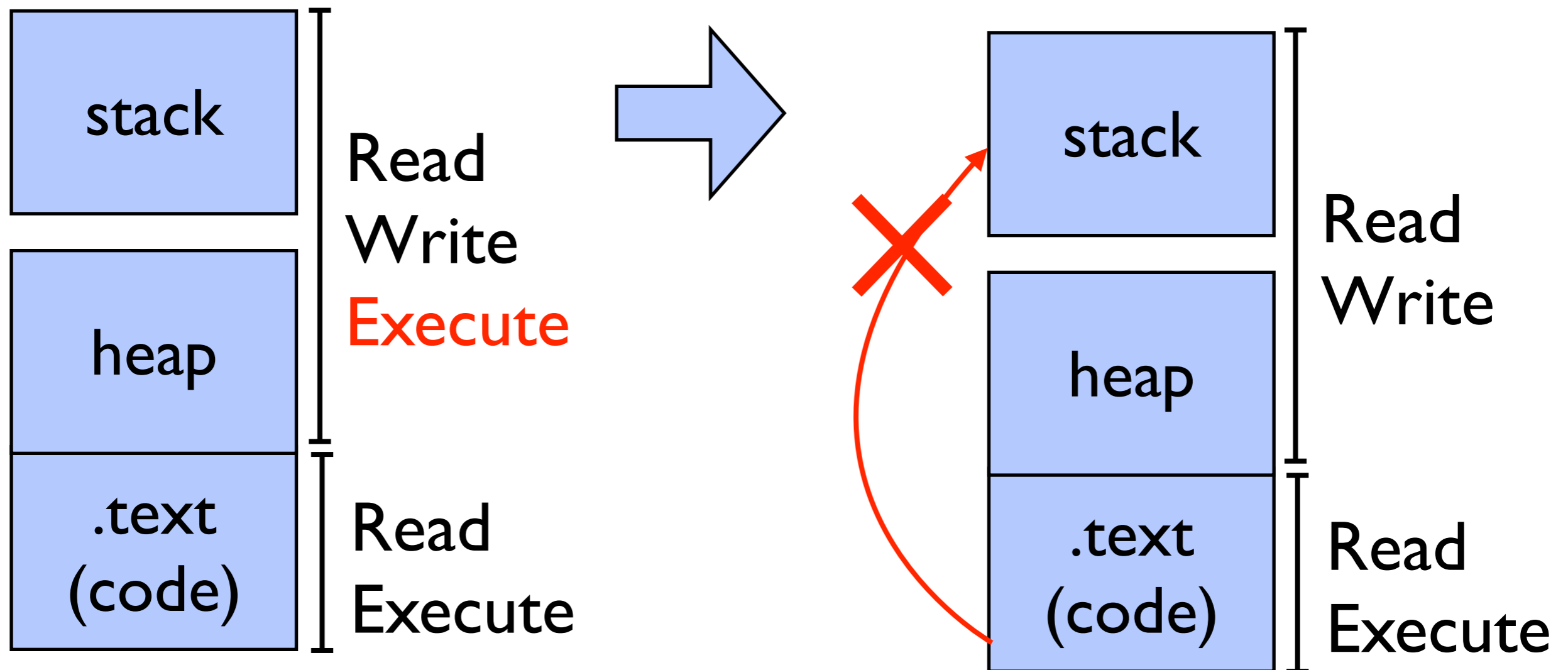
```
snrllcode.  
dup2(sock, 0);  
dup2(sock, 1);  
execve("/bin/sh", 0, 0);  
}
```

Stack:

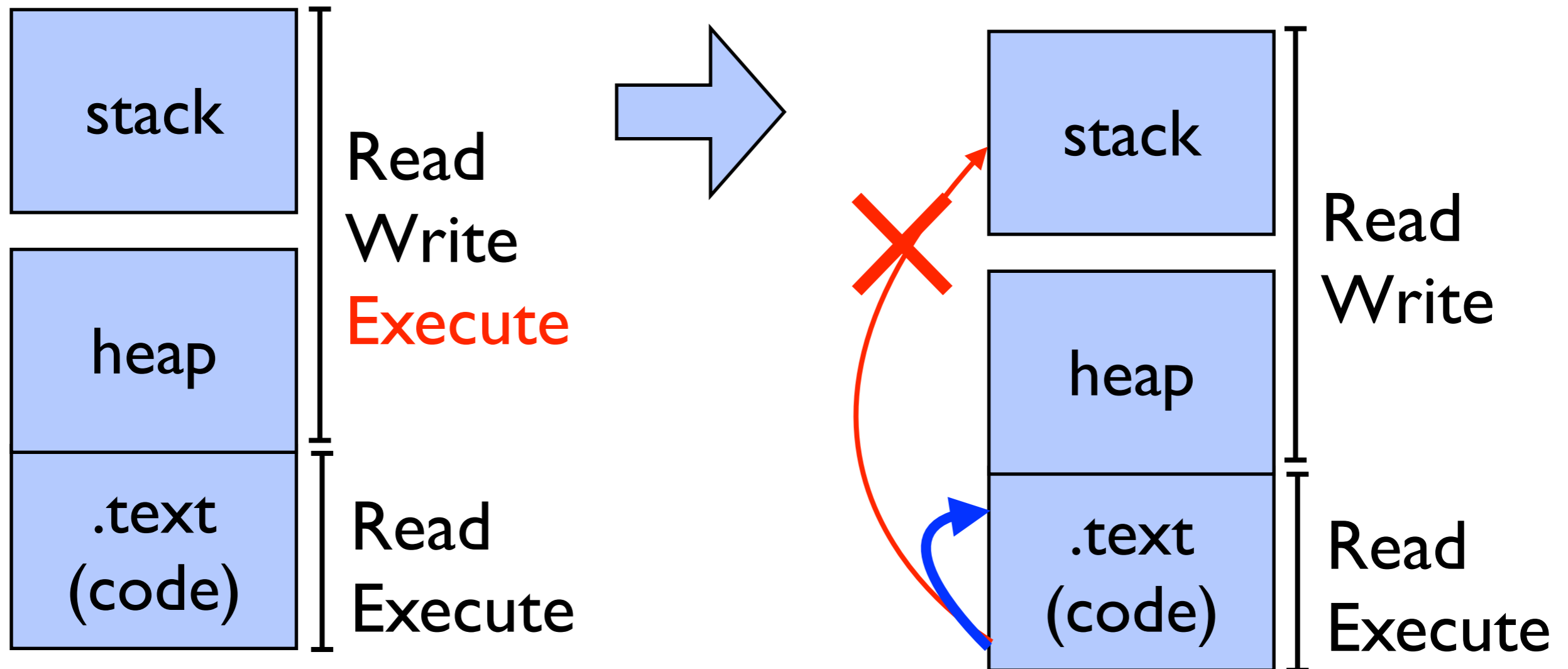
return address
0x600000

0x1029827189
123781923719
823719287319
879181823828

Non-eXecutable (NX) memory

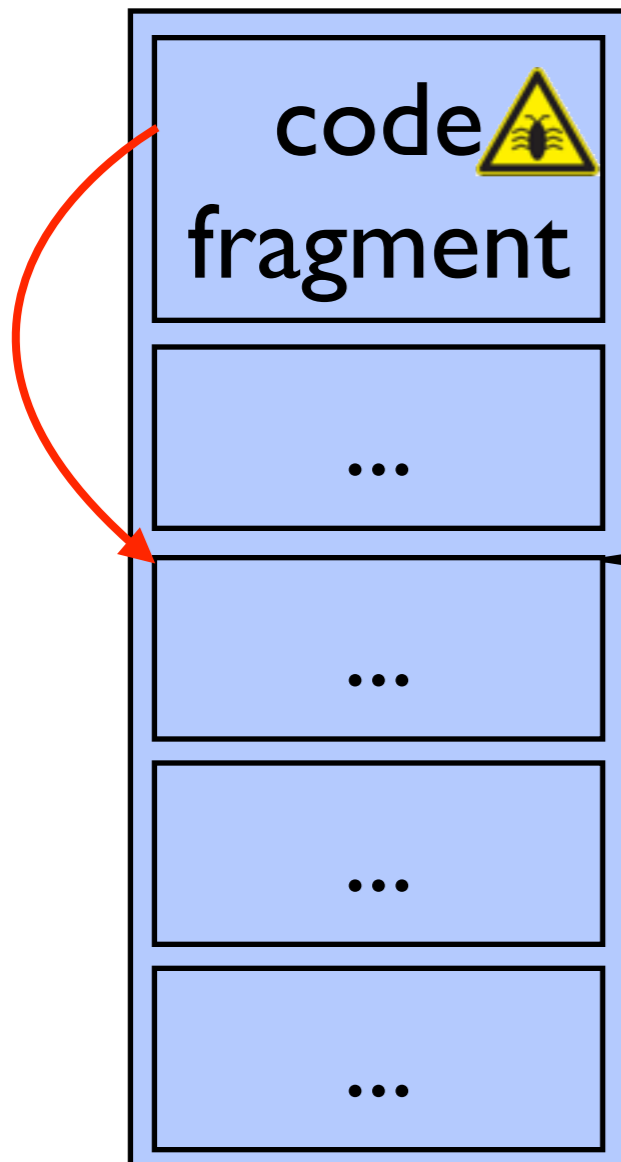


Non-eXecutable (NX) memory



Return-Oriented Programming (ROP)

.text:



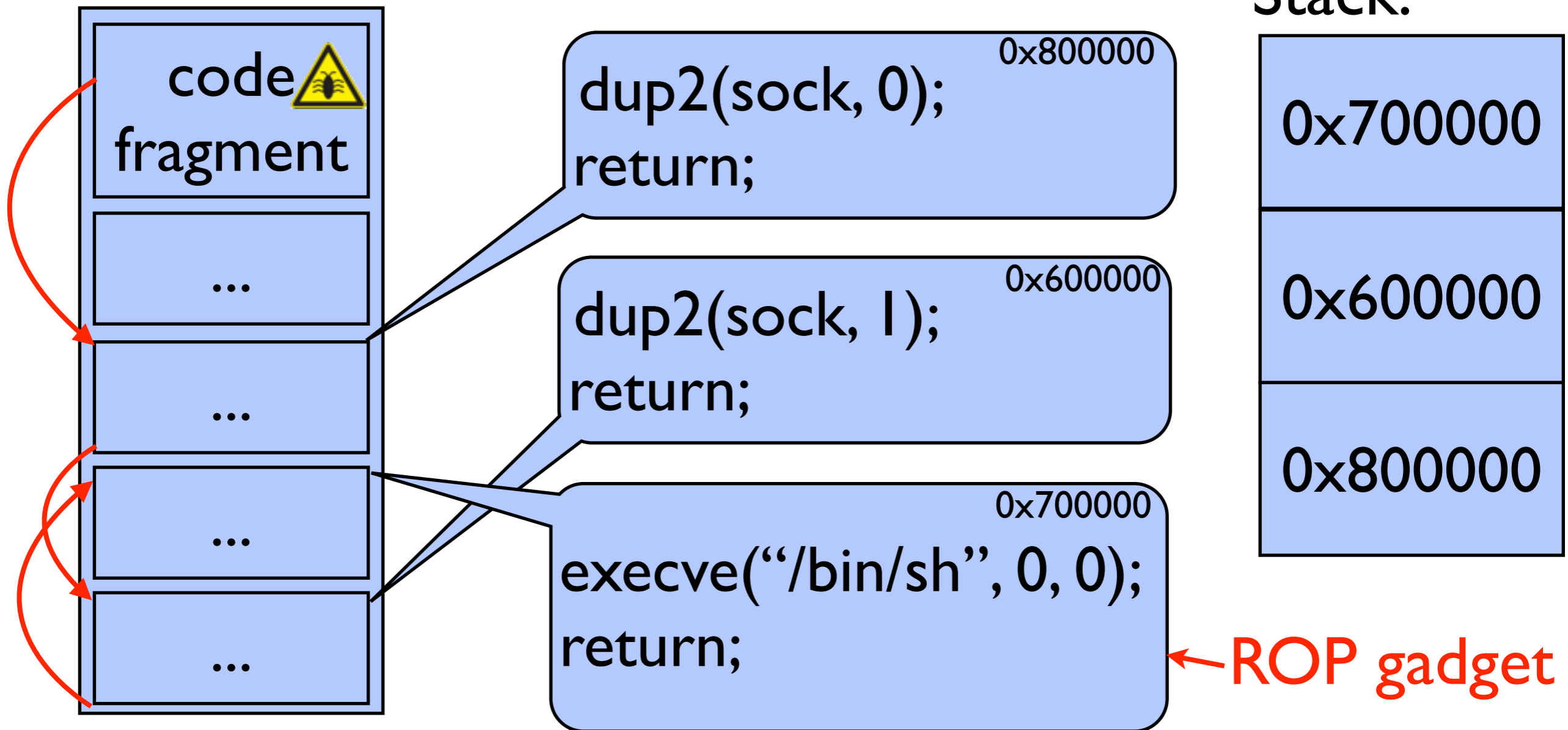
```
dup2(sock, 0);  
dup2(sock, 1);  
execve("/bin/sh", 0, 0);
```

Stack:

0x800000

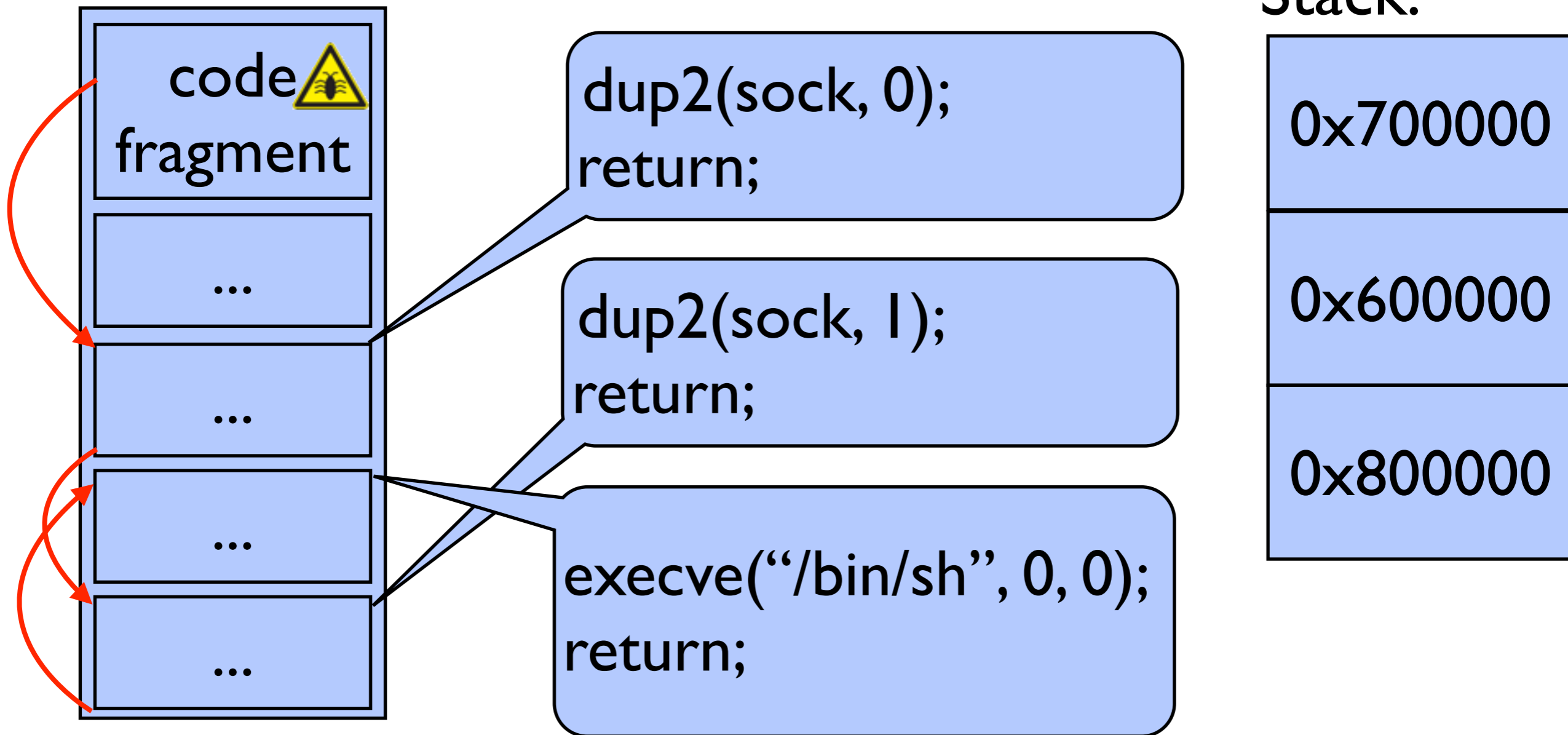
Return-Oriented Programming (ROP)

.text:



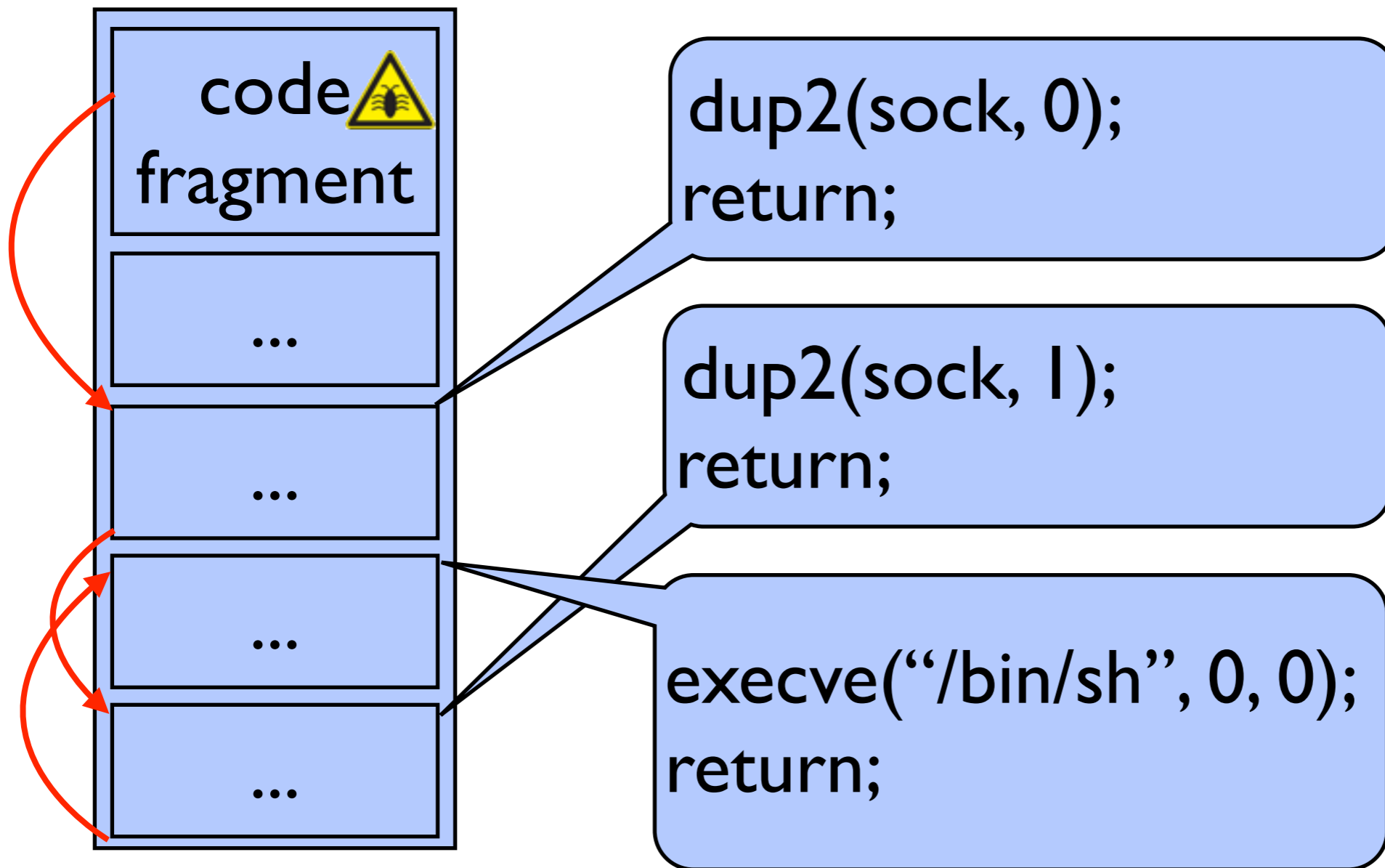
Address Space Layout Randomization (ASLR)

.text: 0x400000

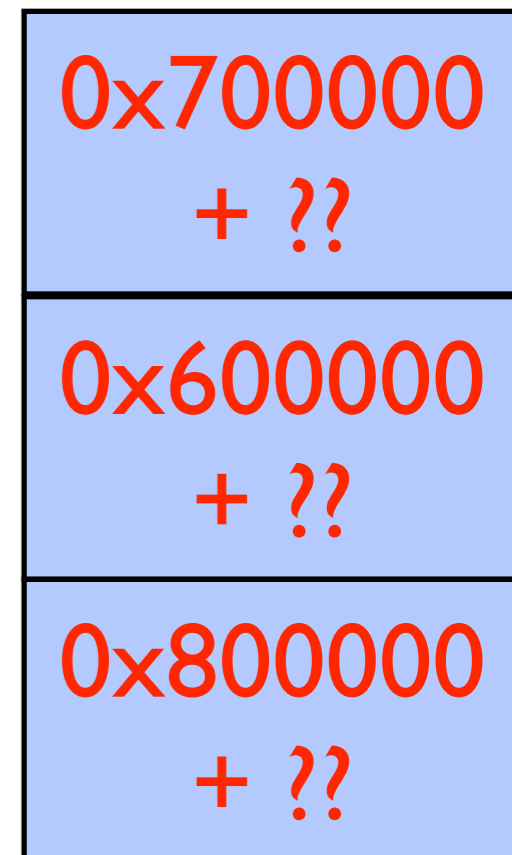


Address Space Layout Randomization (ASLR)

.text: 0x400000 + ??



Stack:



Exploit requirements today

1. Break ASLR.
2. Copy of binary (find ROP gadgets / break NX).
 - Is it even possible to hack unknown applications?

Blind Return-Oriented Programming (BRROP)

1. Break ASLR.
2. Leak binary:
 - Remotely find enough gadgets to call `write()`.
 - `write()` binary from memory to network to disassemble and find more gadgets to finish off exploit.

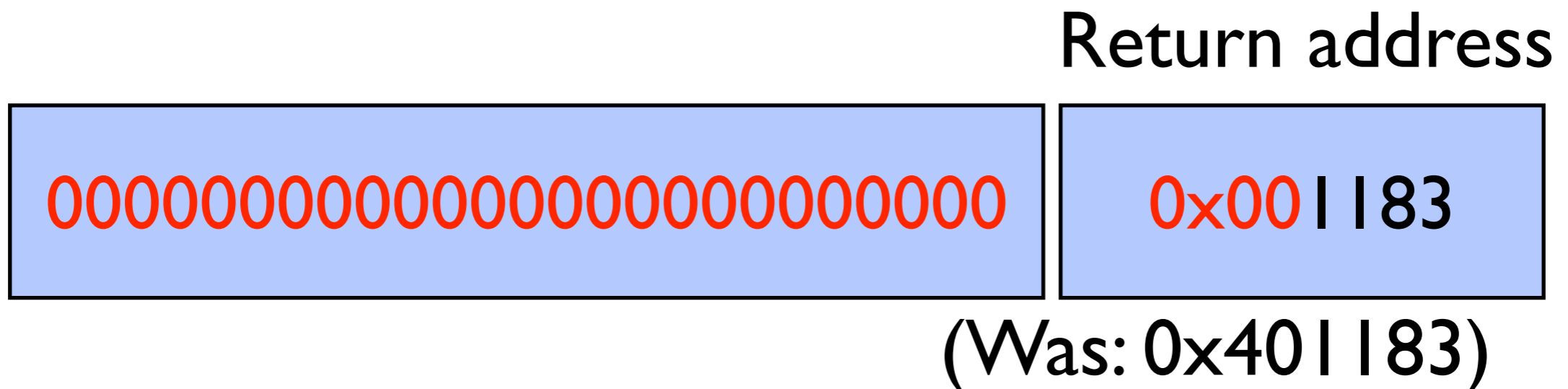
Defeating ASLR: stack reading

- Overwrite a single byte with value X :
 - No crash: stack had value X .
 - Crash: guess X was incorrect.
- Known technique for leaking canaries.



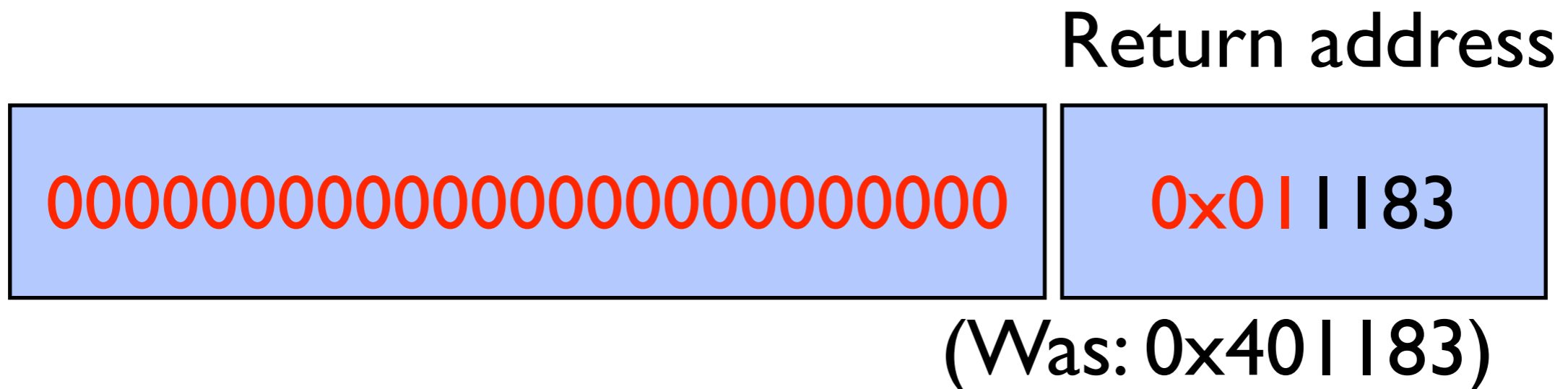
Defeating ASLR: stack reading

- Overwrite a single byte with value X:
 - No crash: stack had value X.
 - Crash: guess X was incorrect.
- Known technique for leaking canaries.



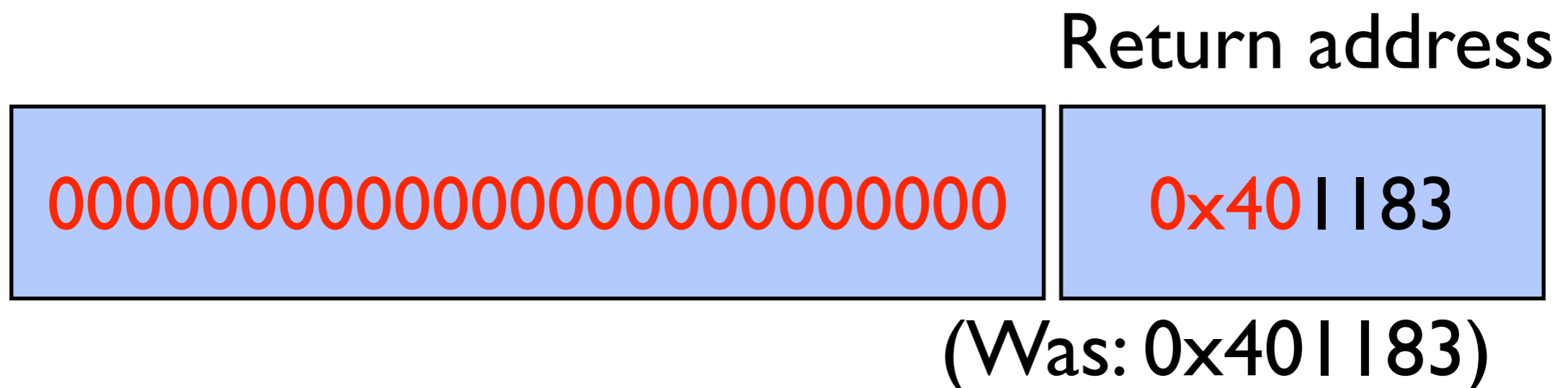
Defeating ASLR: stack reading

- Overwrite a single byte with value X:
 - No crash: stack had value X.
 - Crash: guess X was incorrect.
- Known technique for leaking canaries.



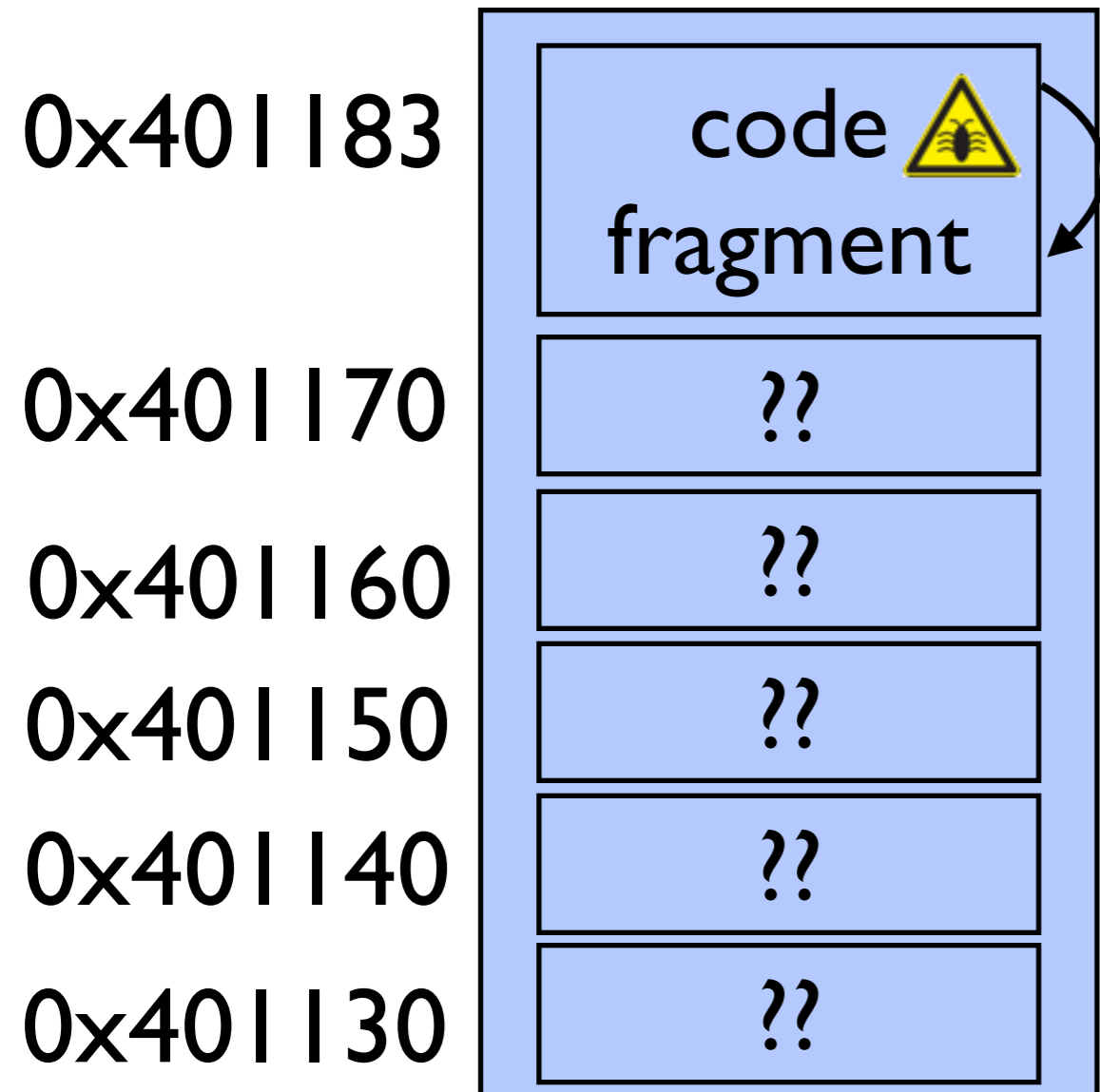
Defeating ASLR: stack reading

- Overwrite a single byte with value X :
 - No crash: stack had value X .
 - Crash: guess X was incorrect.
- Known technique for leaking canaries.

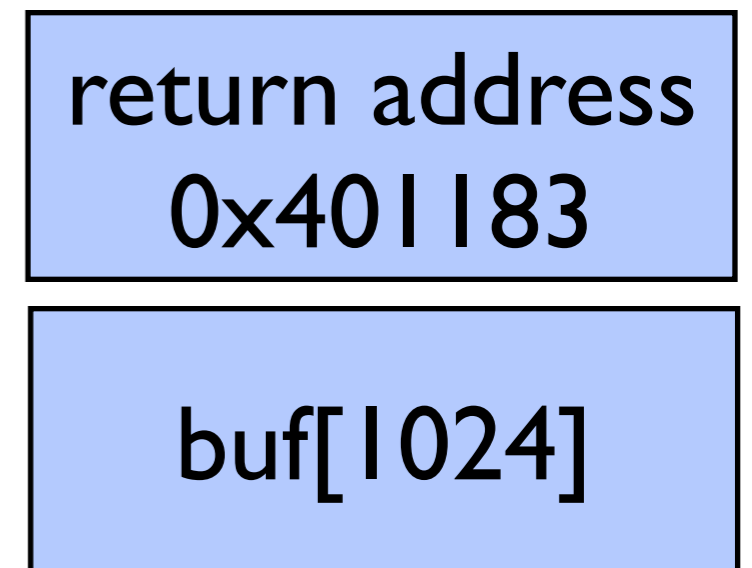


How to find gadgets?

.text:

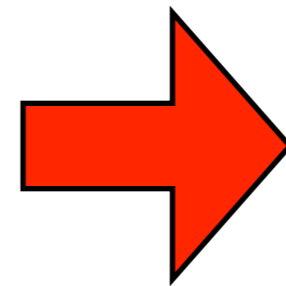
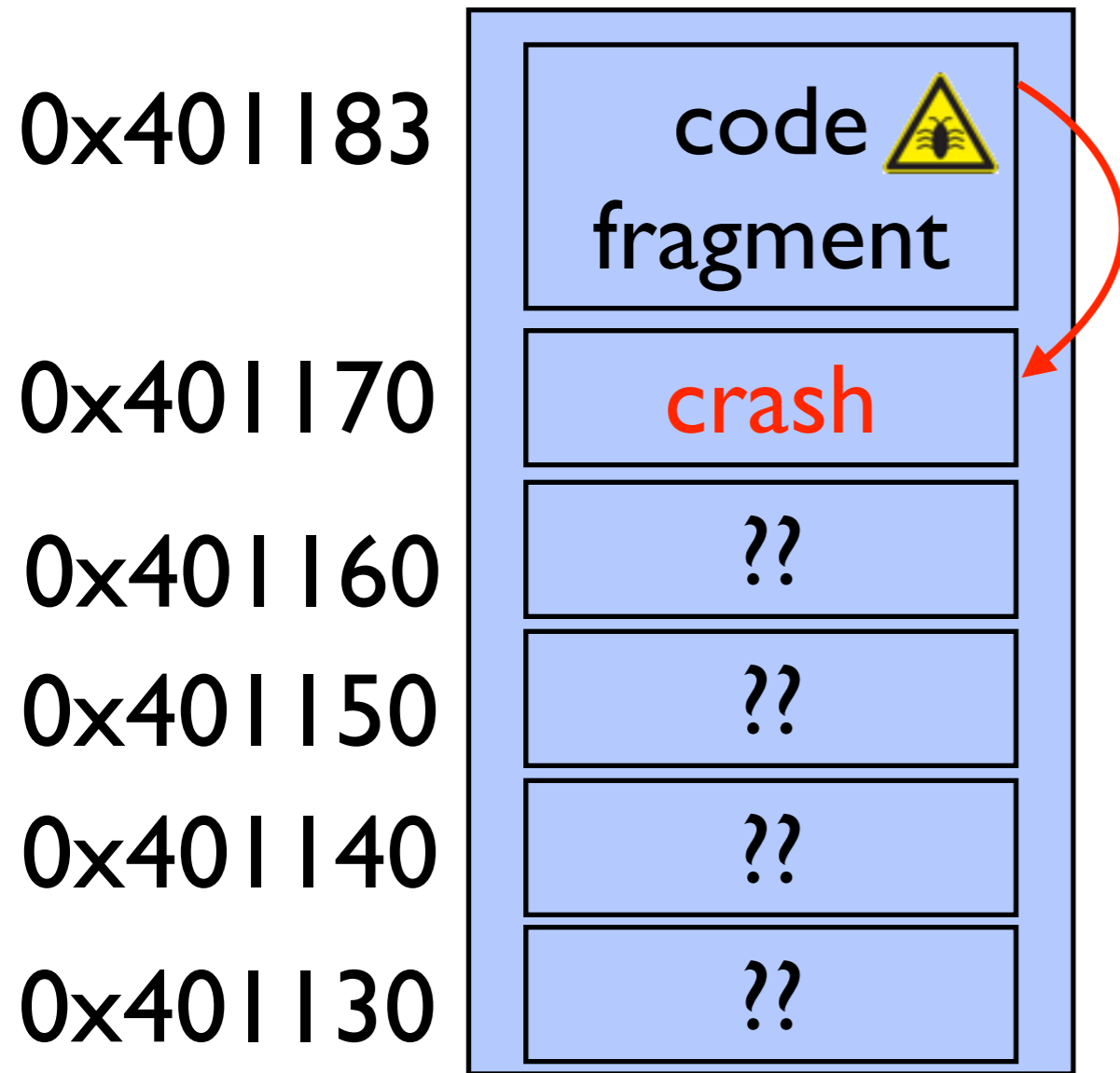


Stack:



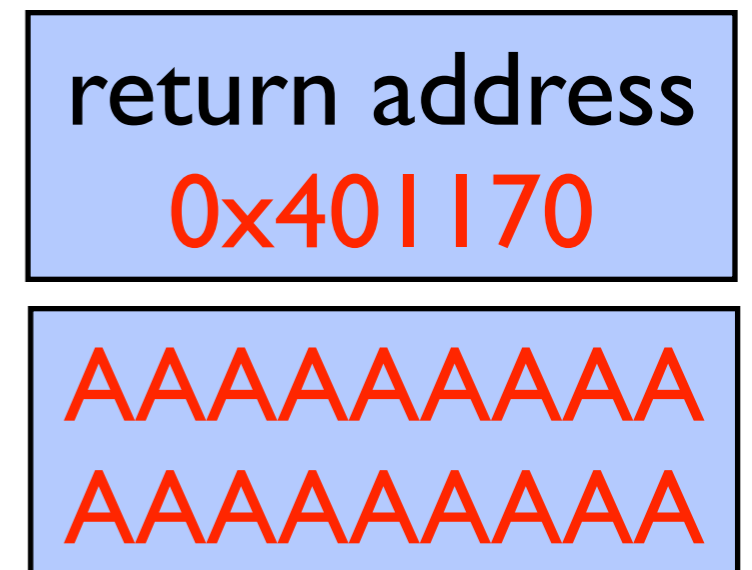
How to find gadgets?

.text:



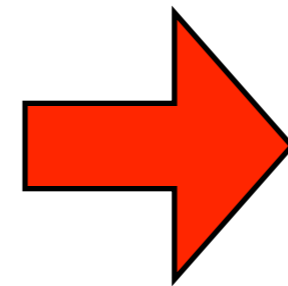
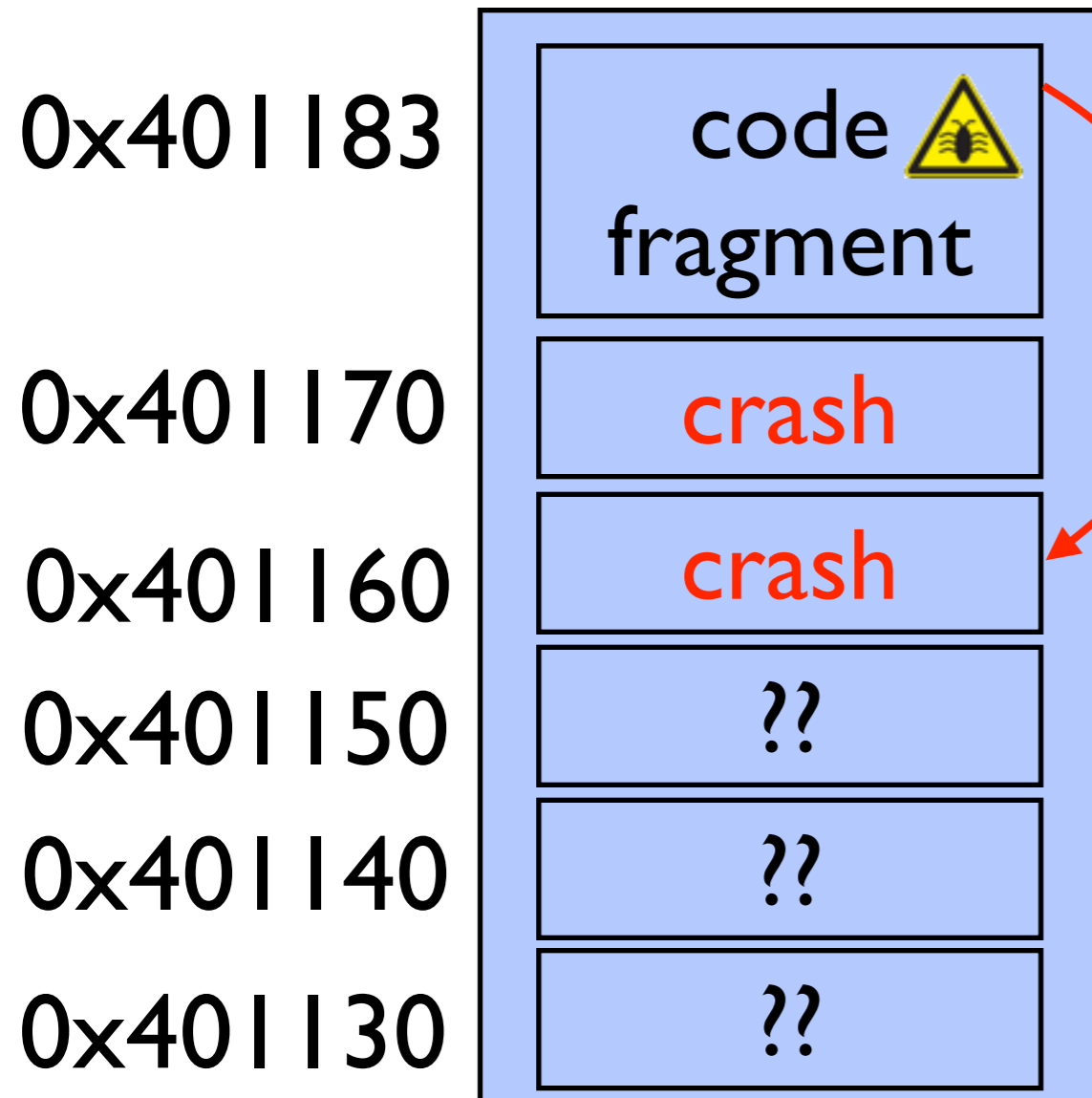
Connection closes

Stack:



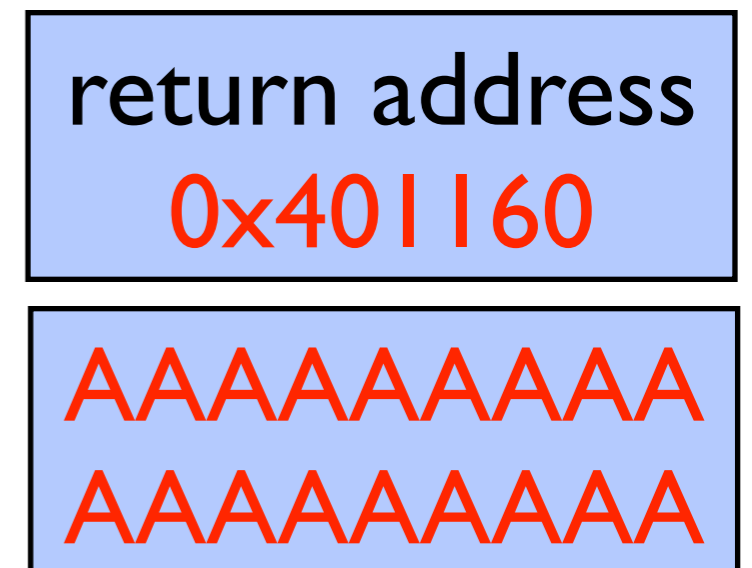
How to find gadgets?

.text:



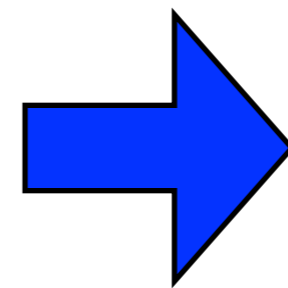
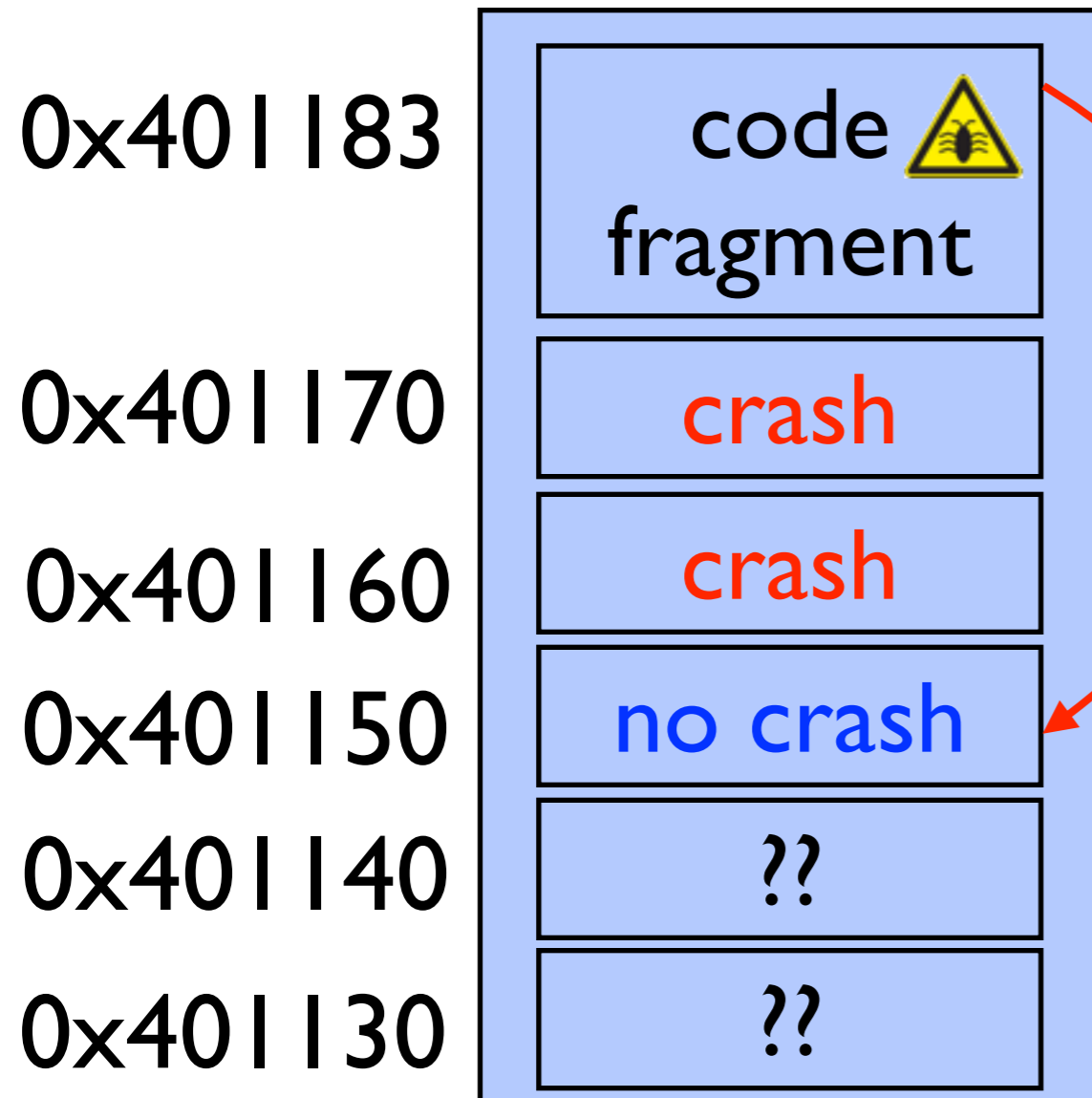
Connection closes

Stack:



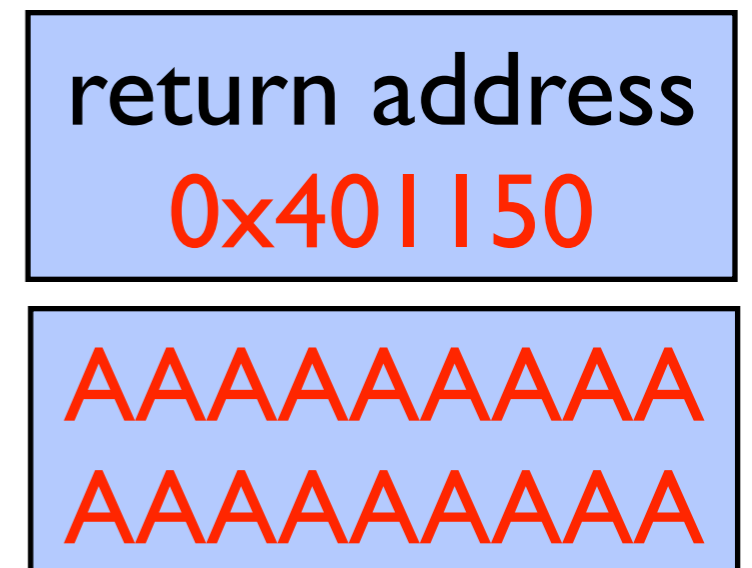
How to find gadgets?

.text:



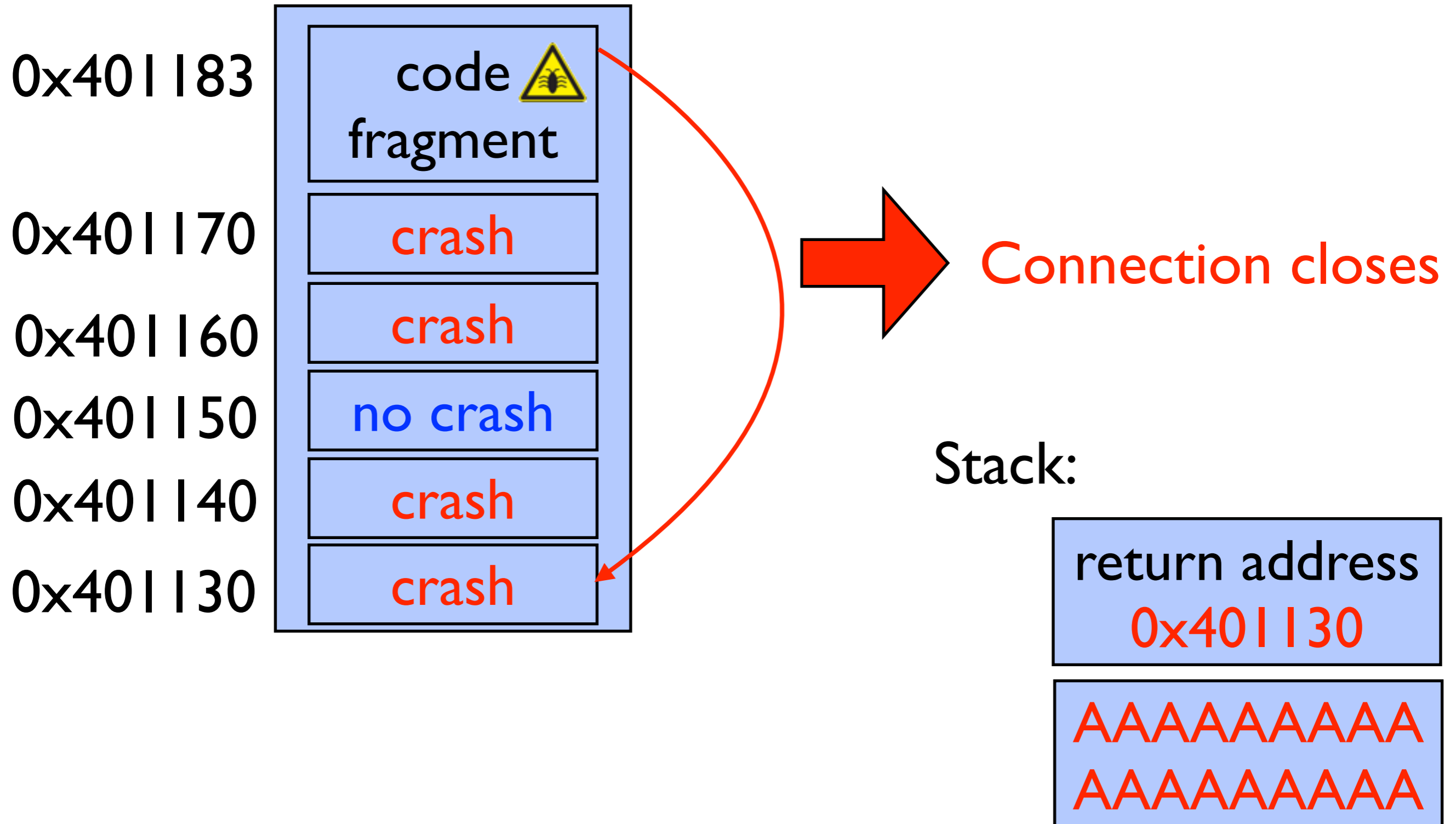
Connection hangs

Stack:



How to find gadgets?

.text:



Three types of gadgets

Stop gadget

```
sleep(10);  
return;
```

- Never crashes

Crash gadget

```
abort();  
return;
```

- Always crashes

Useful gadget

```
dup2(sock, 0);  
return;
```

- Crash depends on return

Three types of gadgets

Stop gadget

```
sleep(10);  
return;
```

- Never crashes

Crash gadget

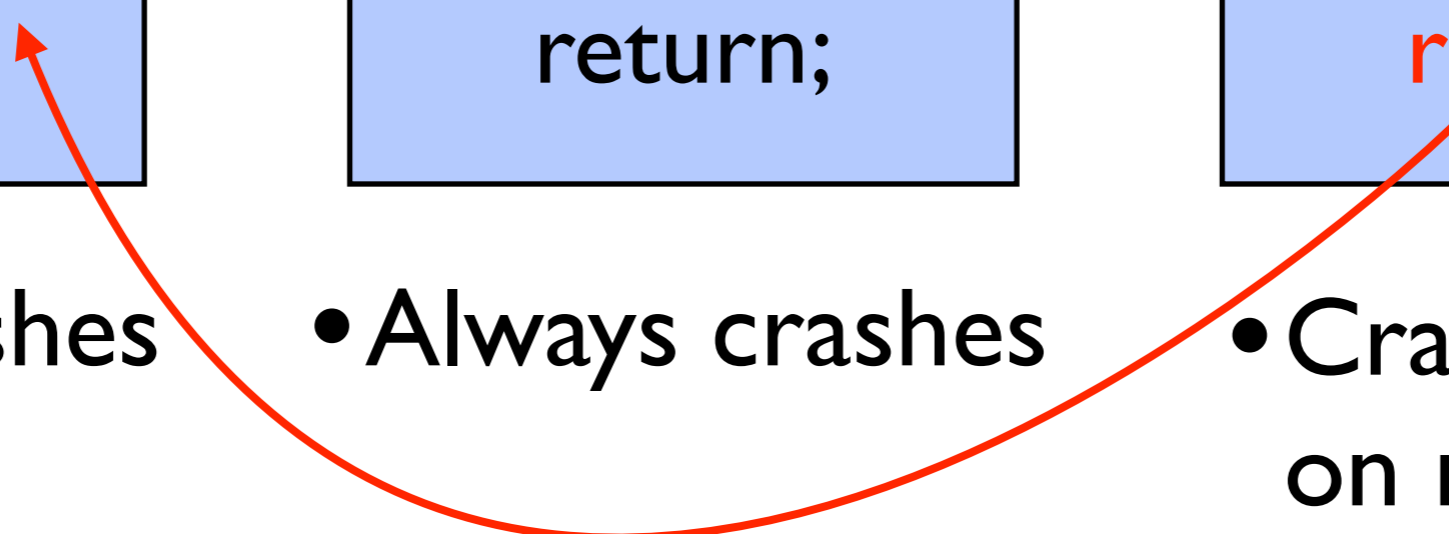
```
abort();  
return;
```

- Always crashes

Useful gadget

```
dup2(sock, 0);  
return;
```

- Crash depends on return



Finding useful gadgets

0x401170

```
dup2(sock, 0);  
return;
```

Stack:

```
other
```

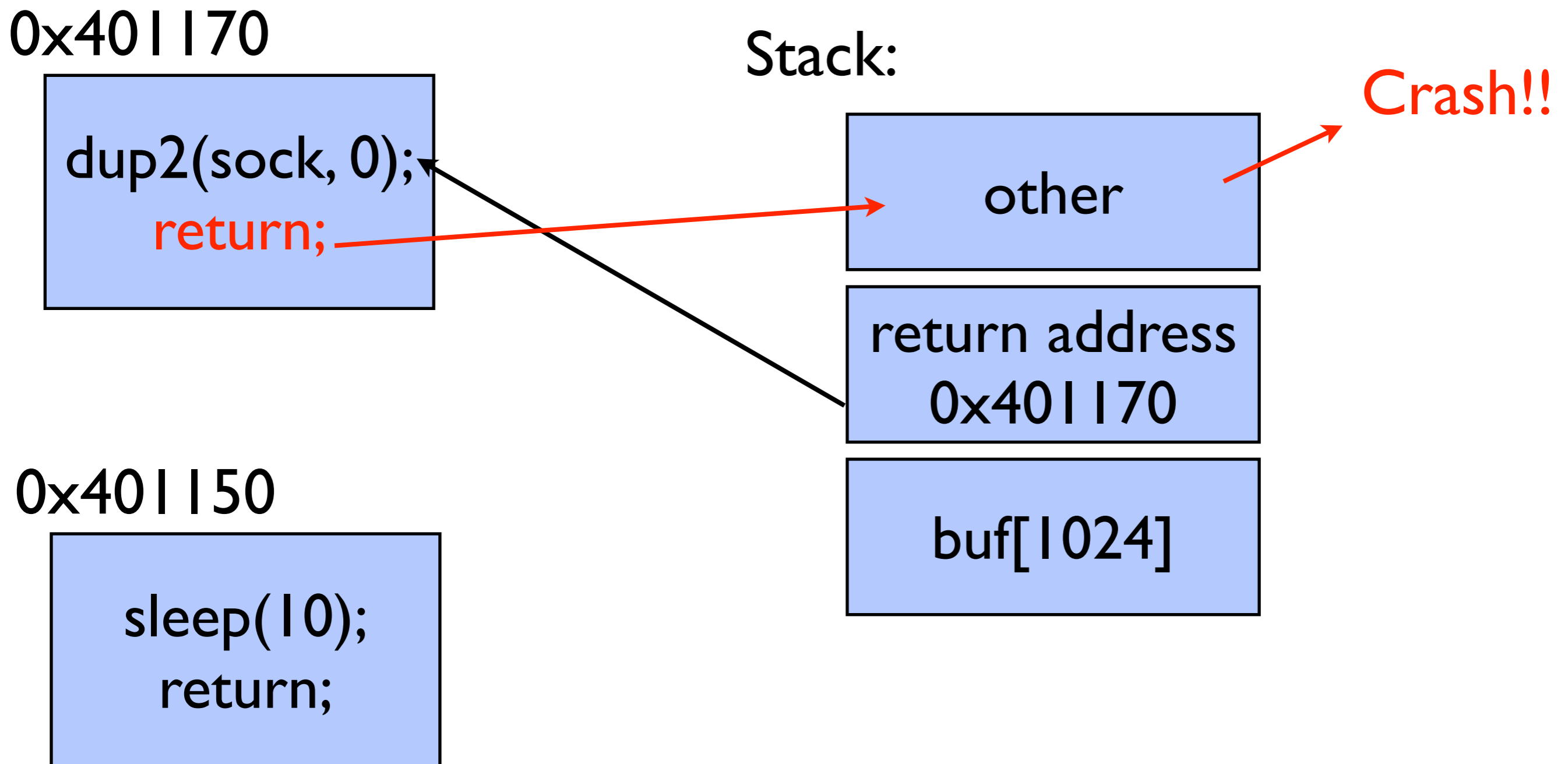
```
return address  
0x401170
```

```
buf[1024]
```

Crash!!

0x401150

```
sleep(10);  
return;
```



Finding useful gadgets

0x401170

```
dup2(sock, 0);  
return;
```

Stack:

```
0x401150
```

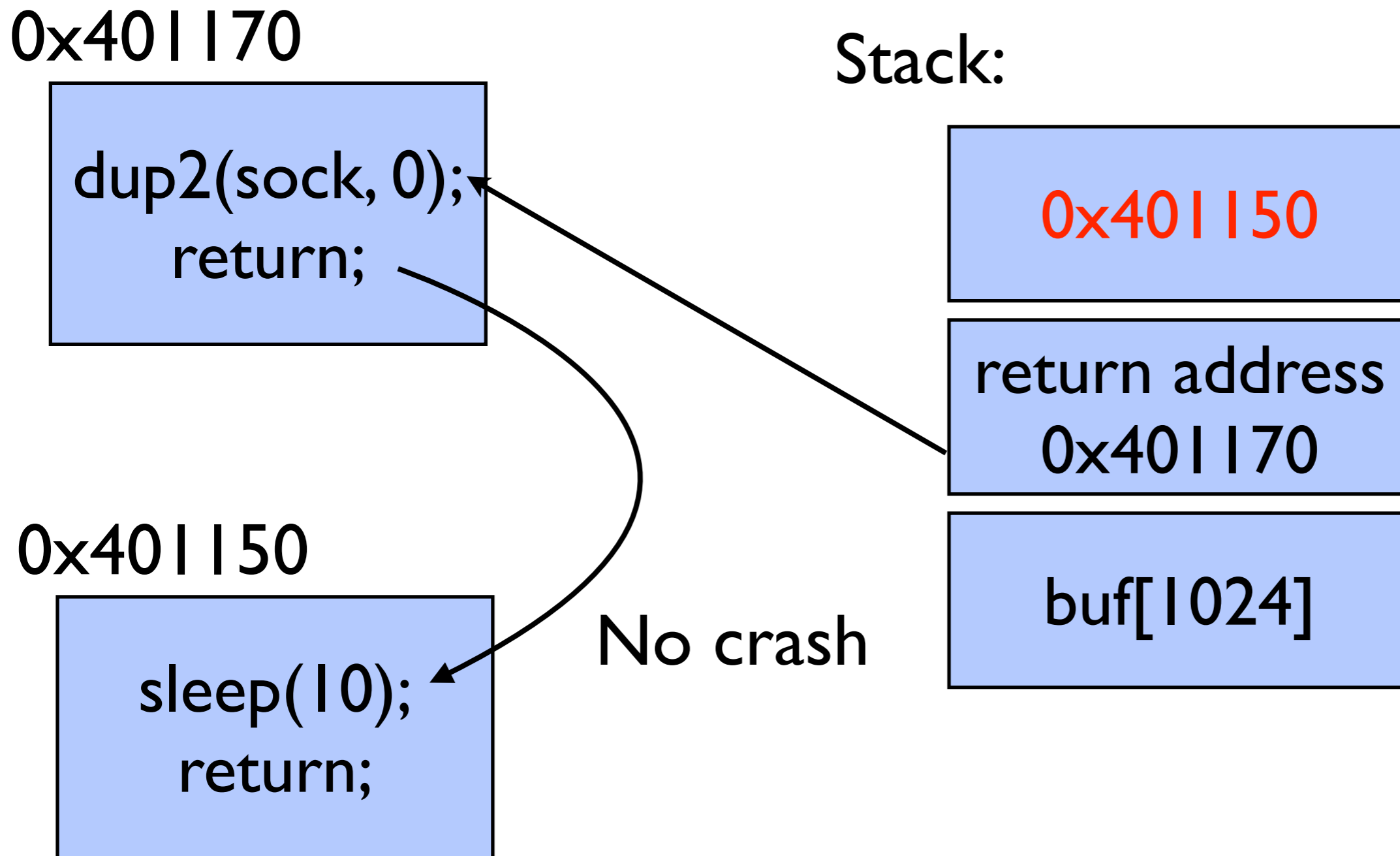
```
return address  
0x401170
```

```
buf[1024]
```

0x401150

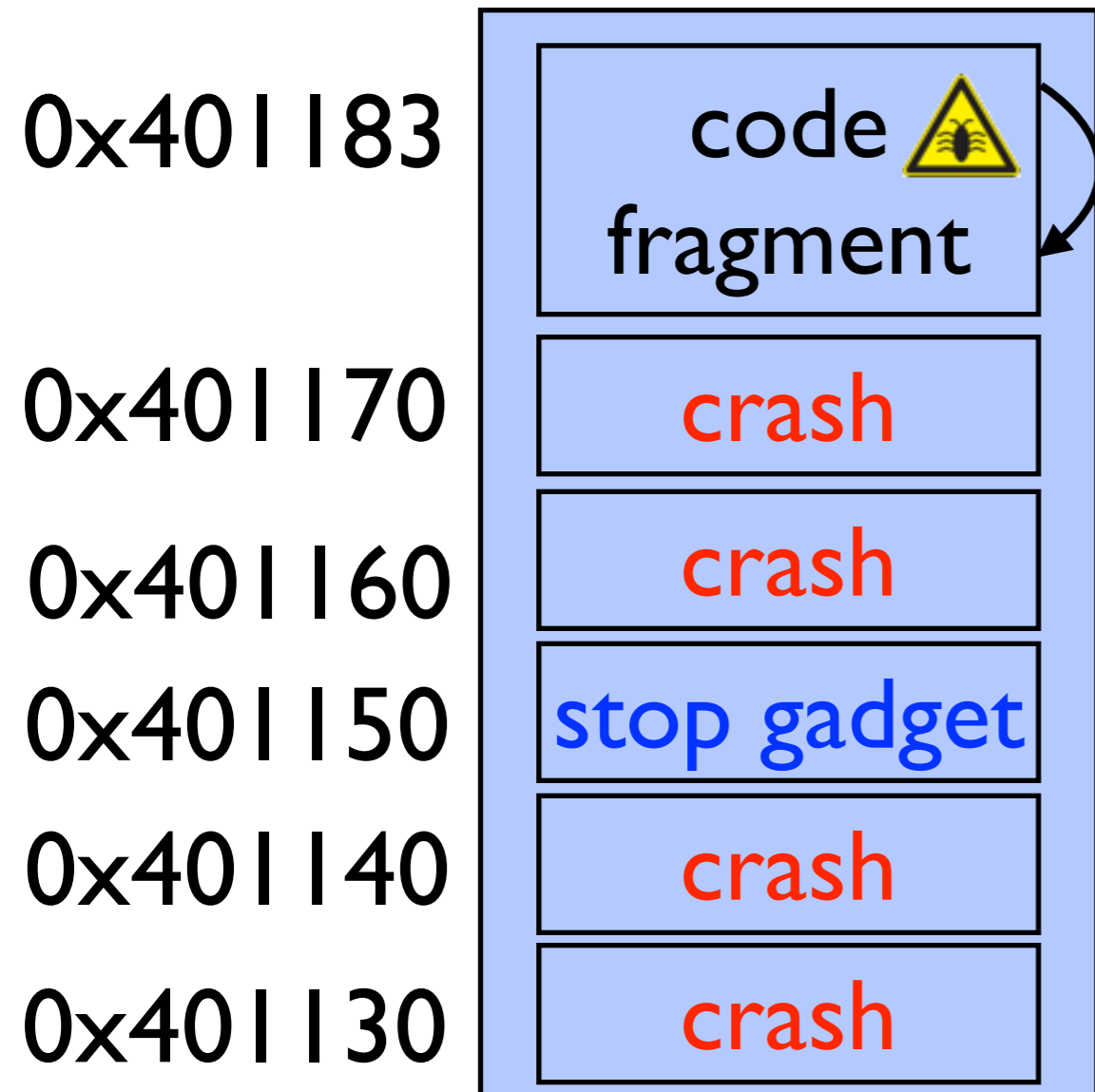
```
sleep(10);  
return;
```

No crash

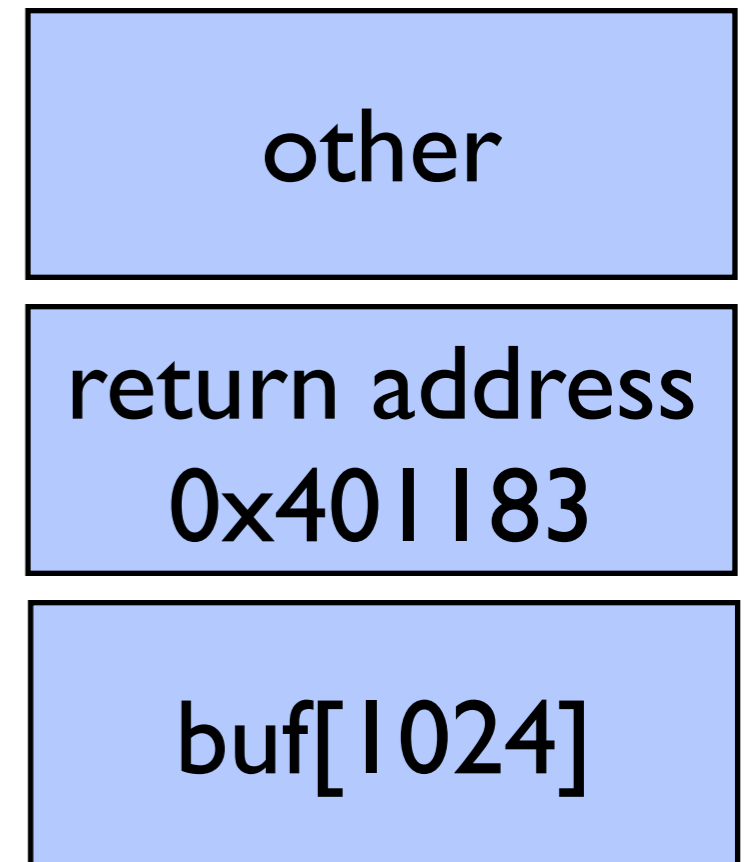


How to find gadgets?

.text:

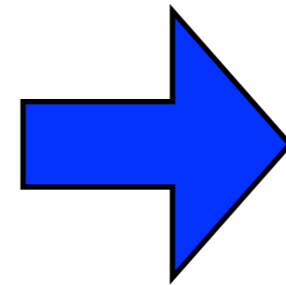
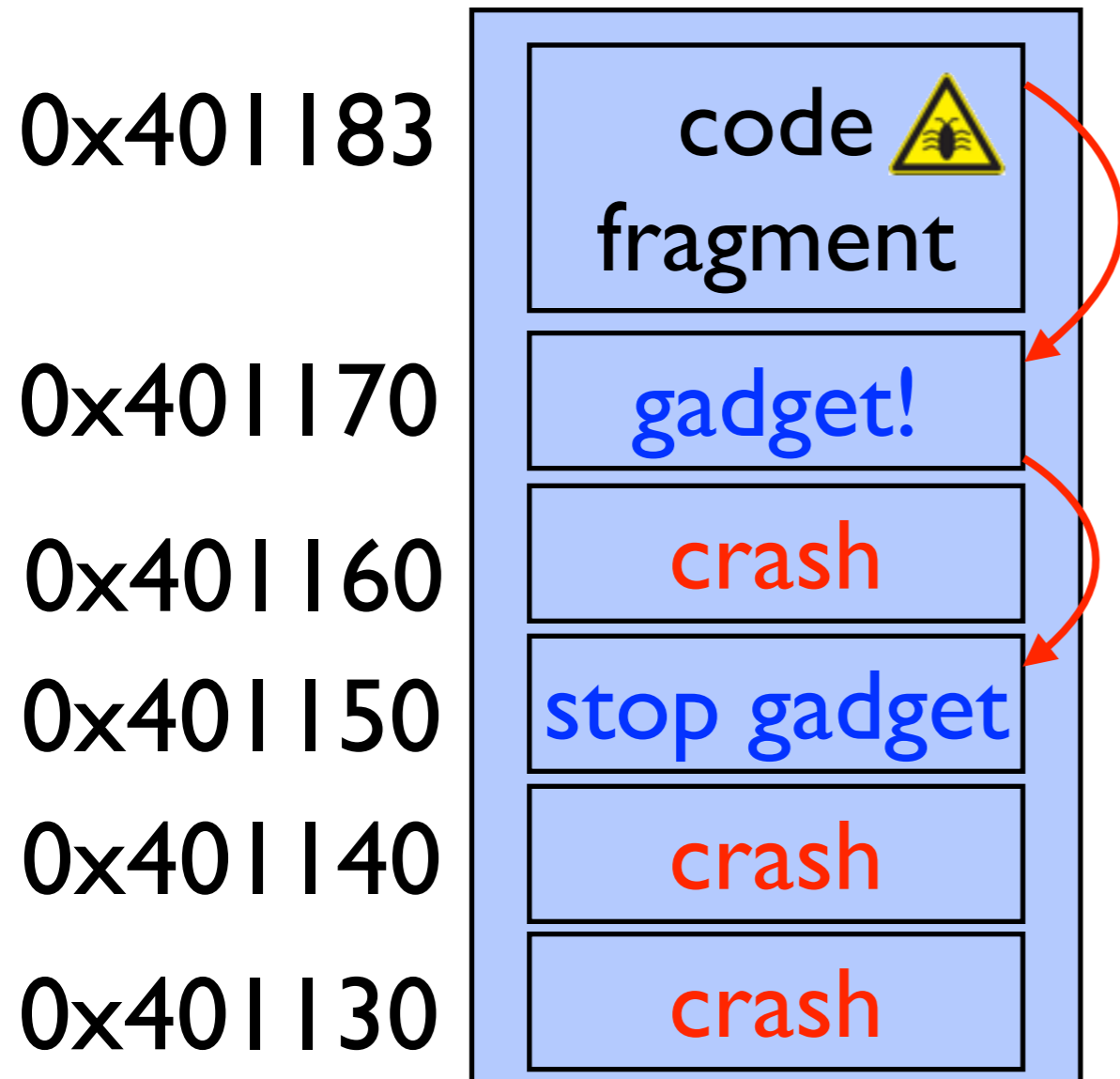


Stack:



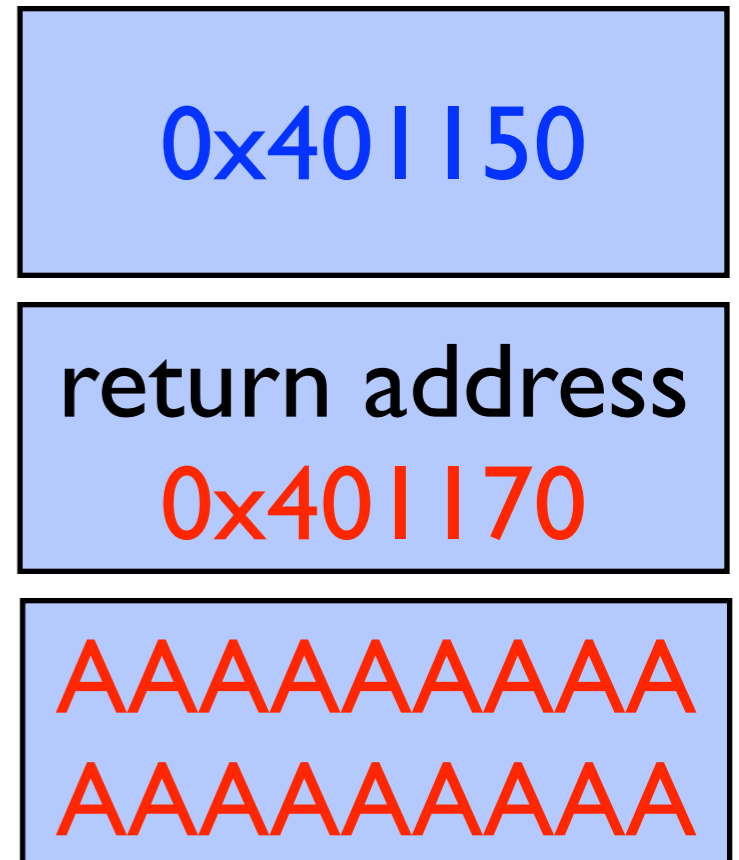
How to find gadgets?

.text:



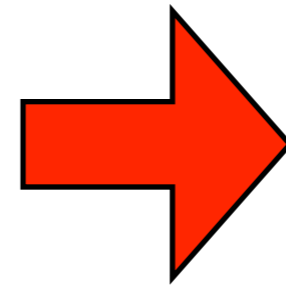
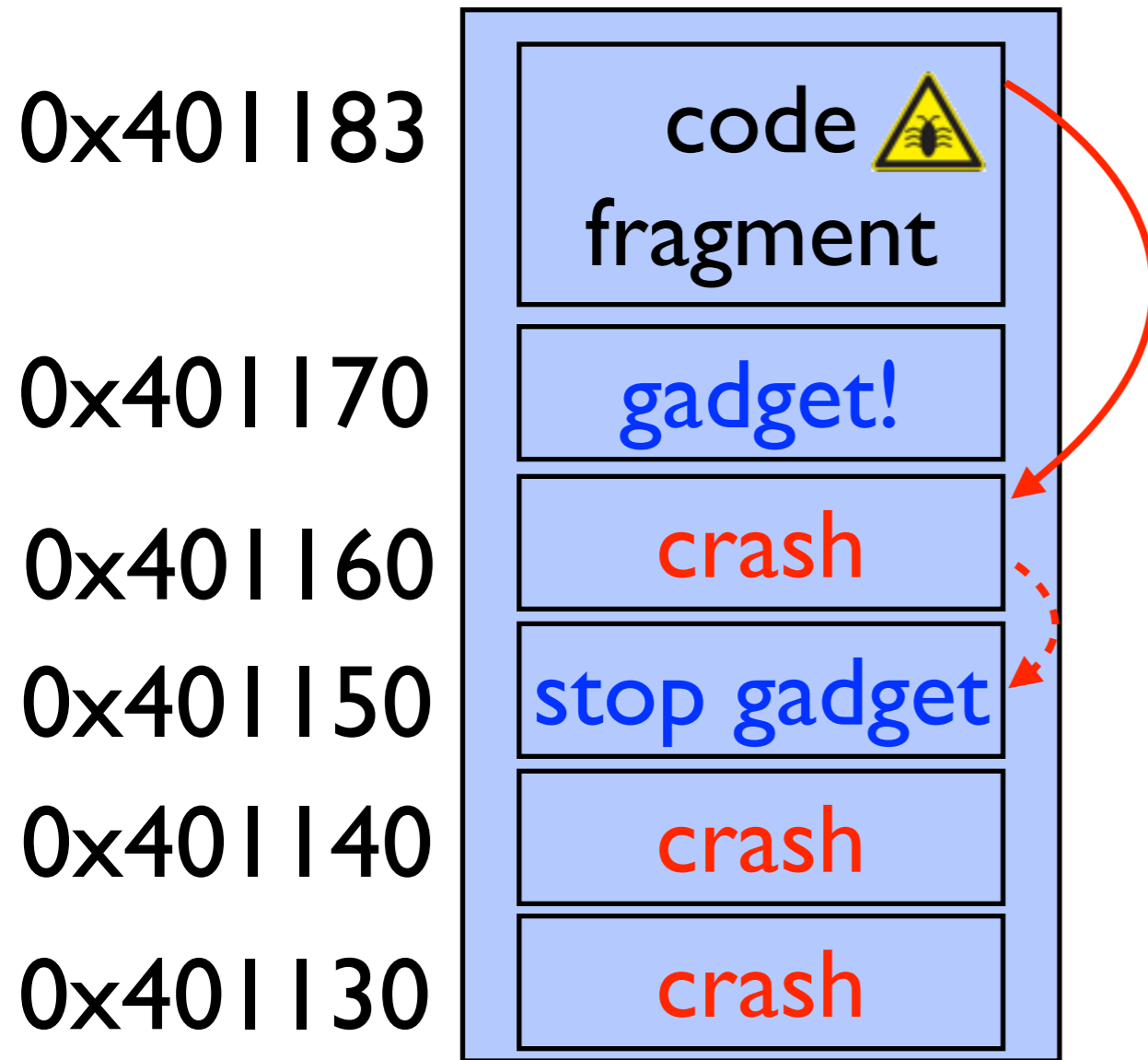
Connection hangs

Stack:



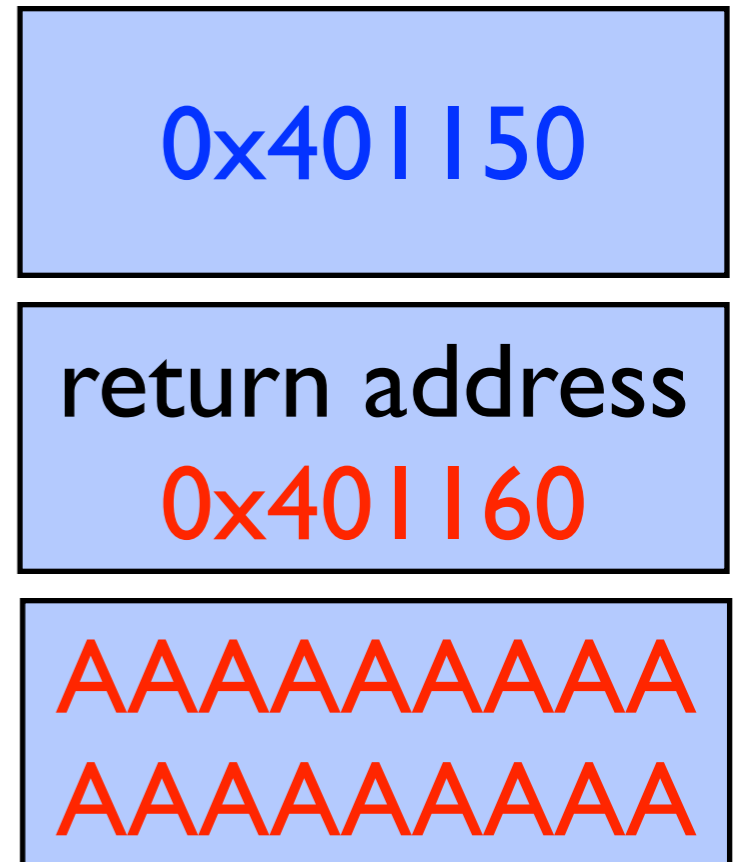
How to find gadgets?

.text:



Connection closes

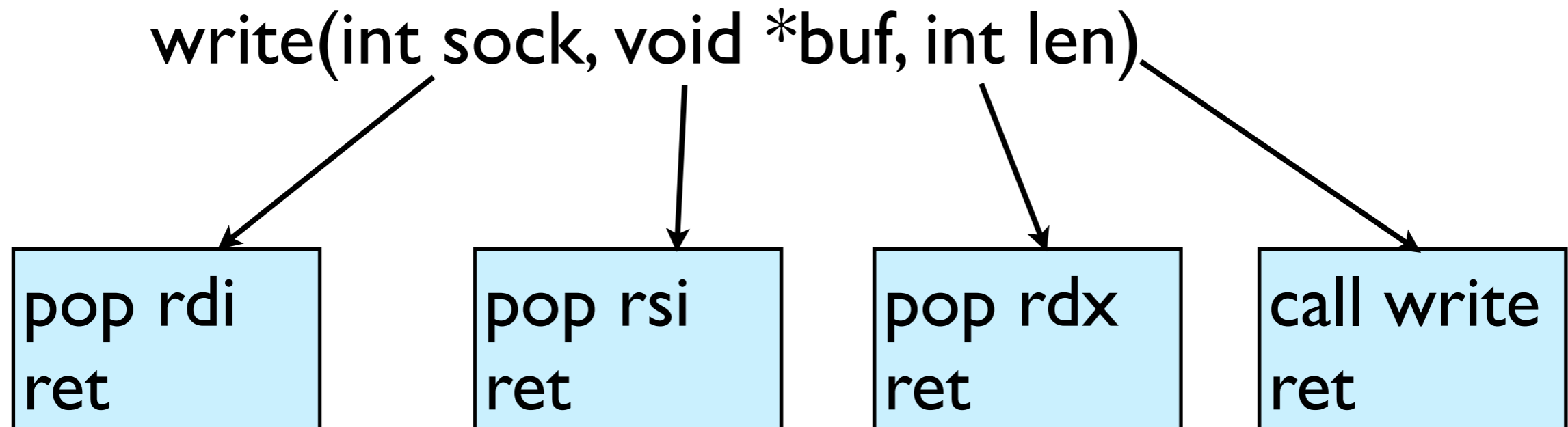
Stack:



What are we looking for?

```
write(int sock, void *buf, int len)
```

What are we looking for?



What are we looking for?

write(int sock, void *buf, int len)

pop rdi
ret

pop rsi
ret

pop rdx
ret

call write
ret

AAAAA
buf[1024]

0x400000
ret addr

sock
rdi

0x500000
buf
rsi

0x600000
len
rdx

0x700000

Pieces of the puzzle

stop gadget
[call sleep]

pop rsi
ret

pop rdi
ret

pop rdx
ret

call write
ret

Pieces of the puzzle

stop gadget
[call sleep]

pop rsi
ret

pop rdi
ret

call strcmp
ret

call write
ret

Pieces of the puzzle

```
pop rsi  
ret
```

```
pop rdi  
ret
```

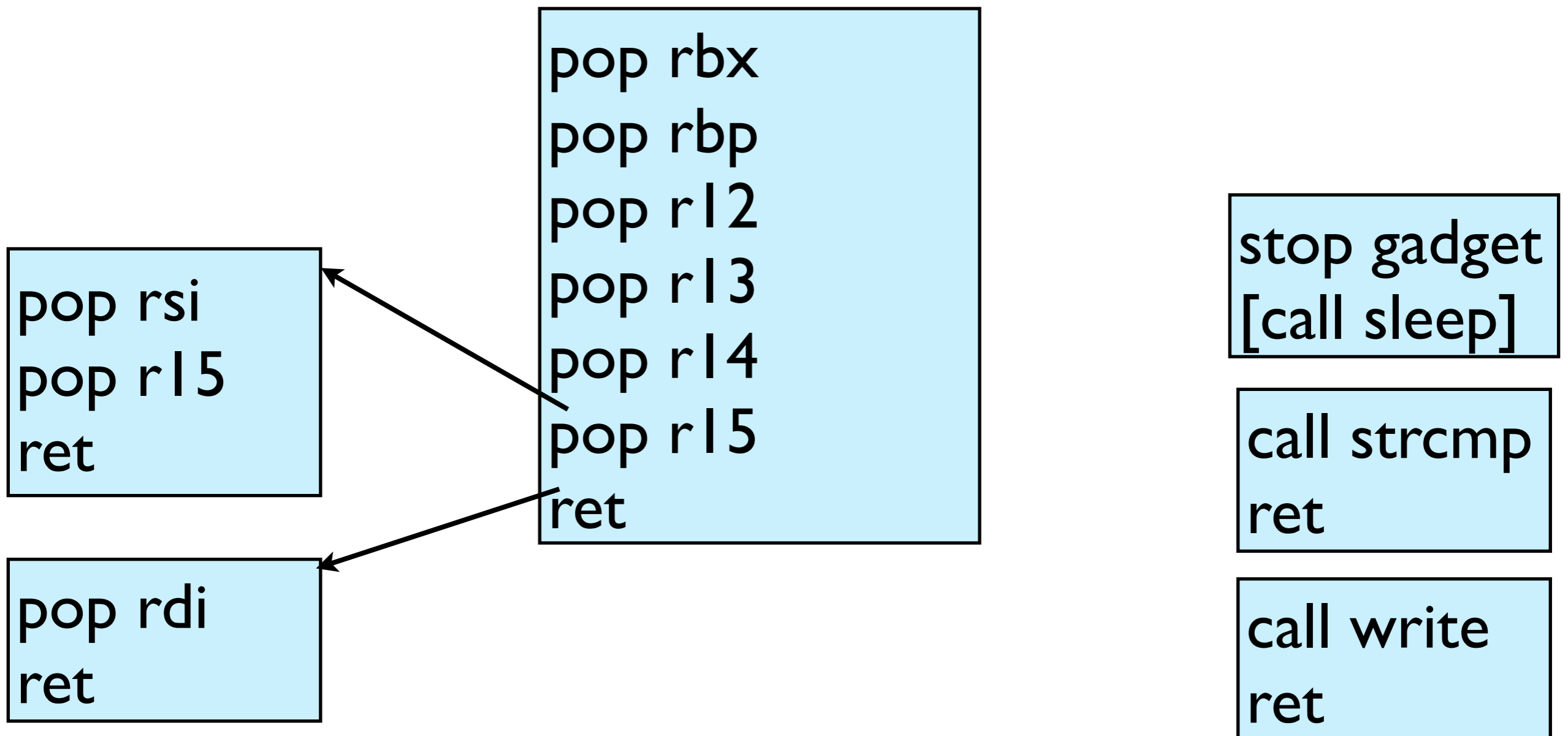
```
stop gadget  
[call sleep]
```

```
call strcmp  
ret
```

```
call write  
ret
```

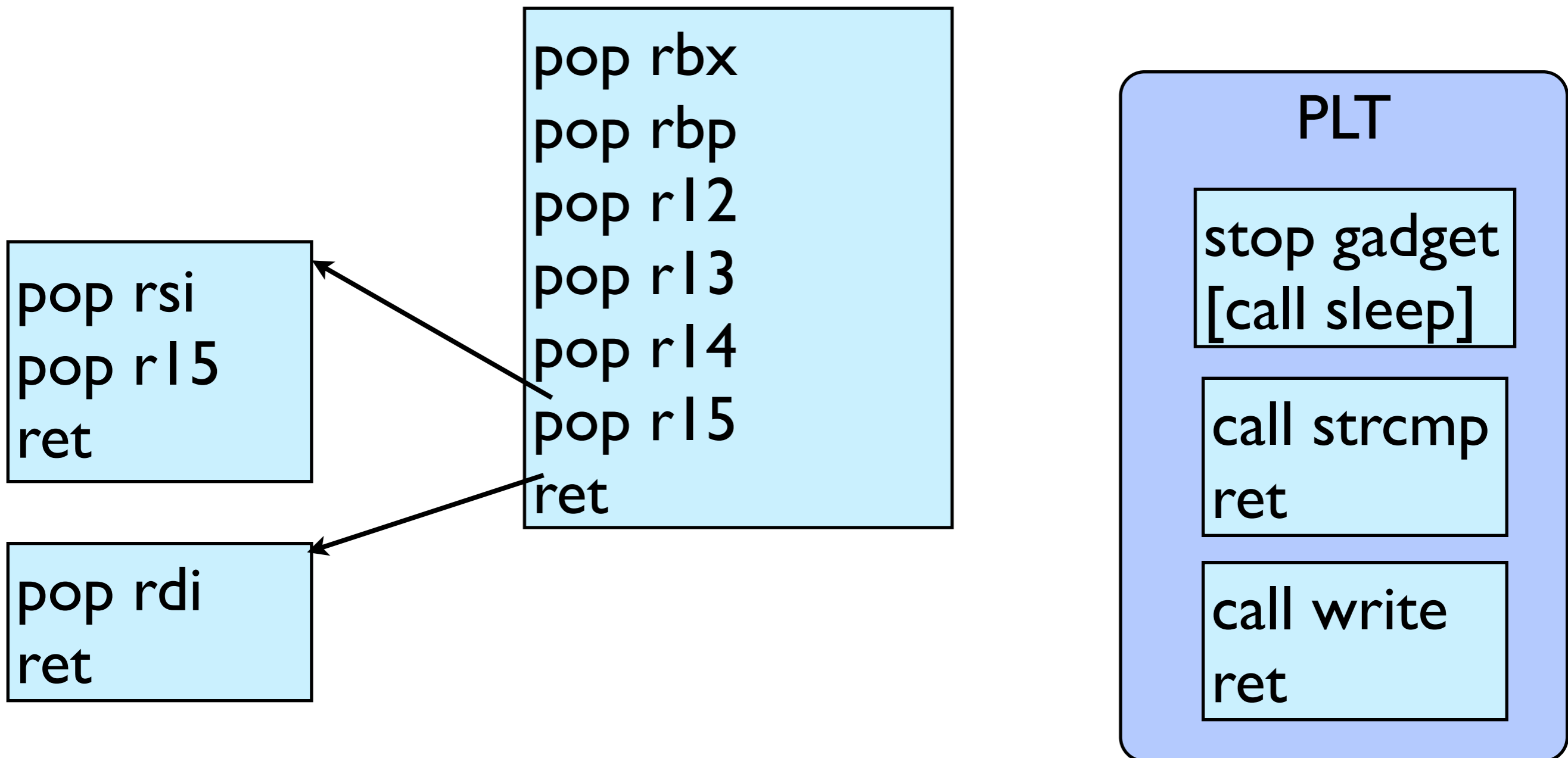
Pieces of the puzzle

The BR0P gadget



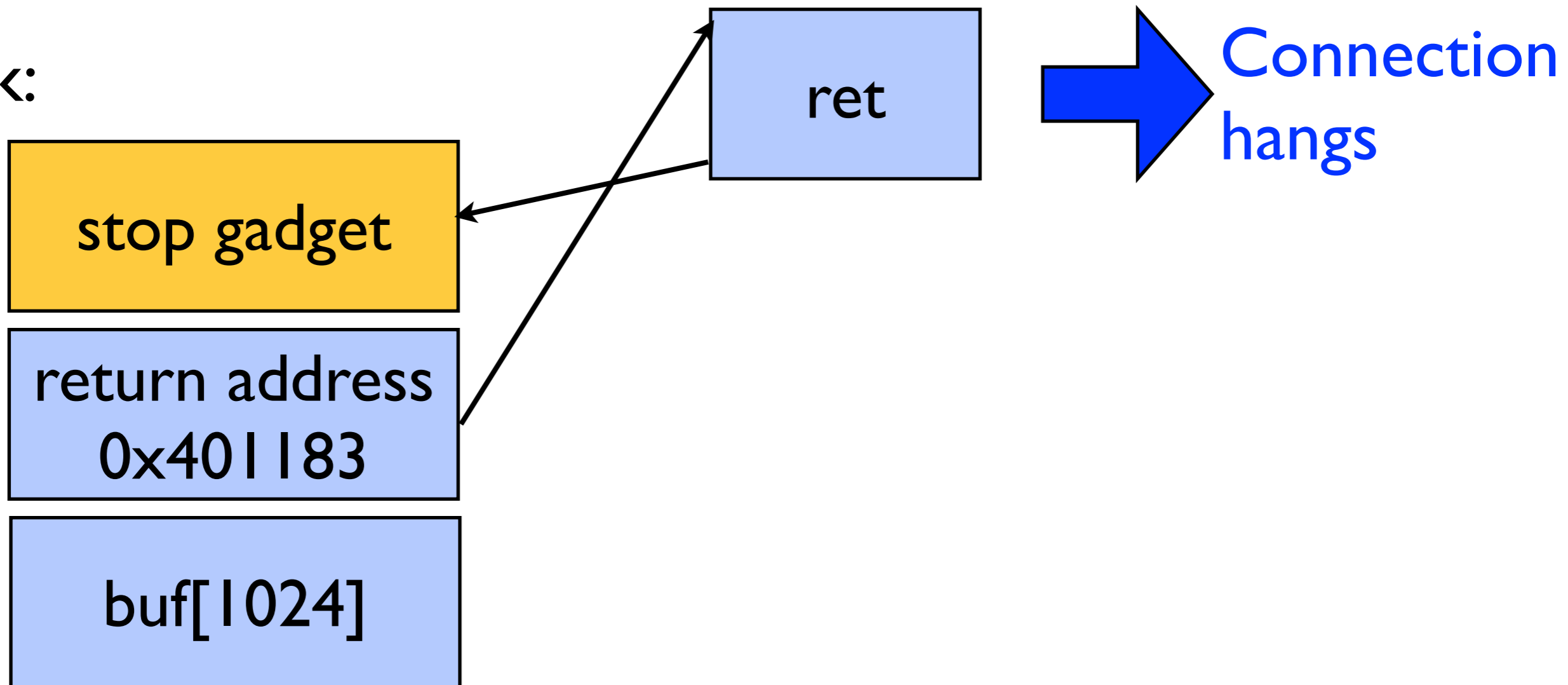
Pieces of the puzzle

The BR0P gadget



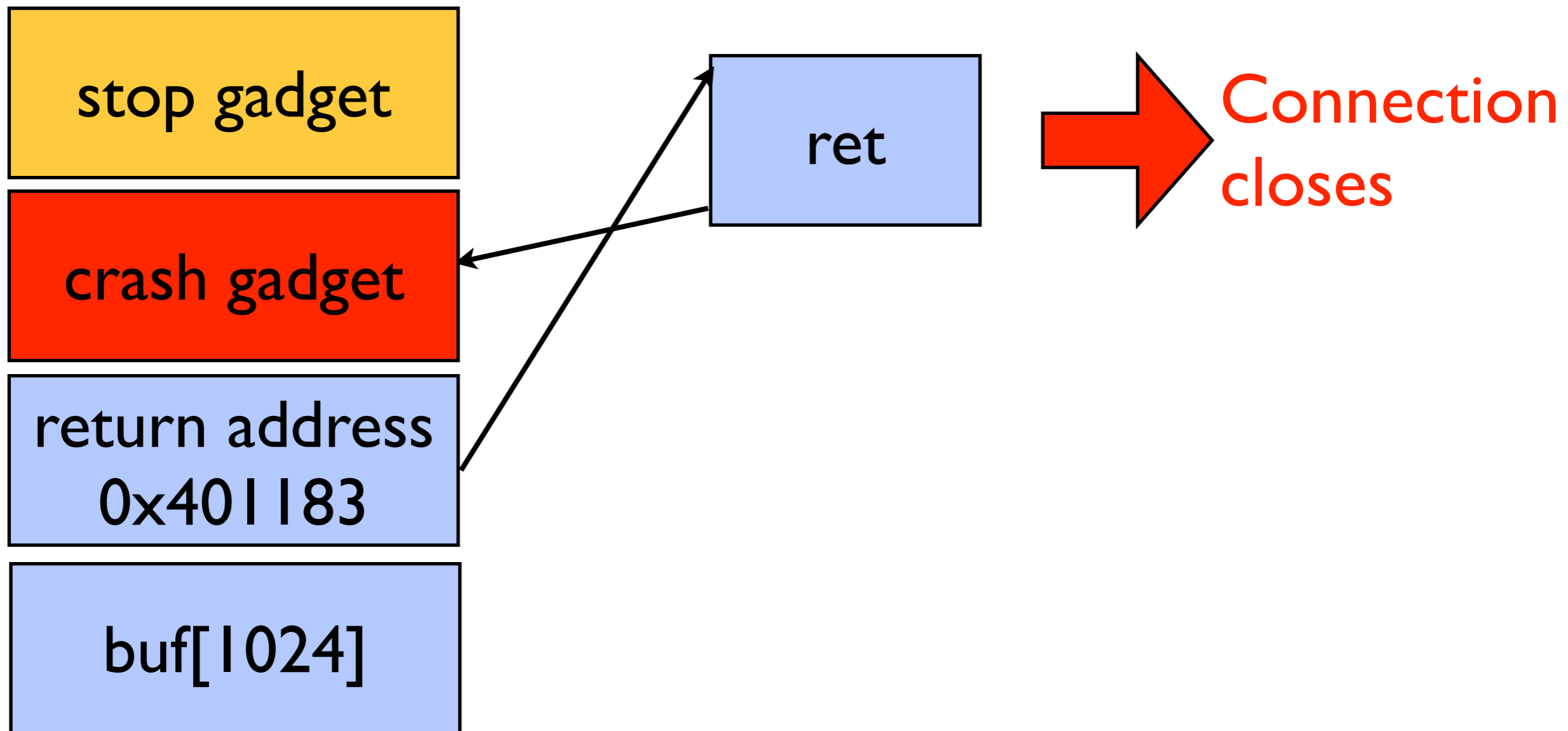
Finding the BROOP gadget

Stack:



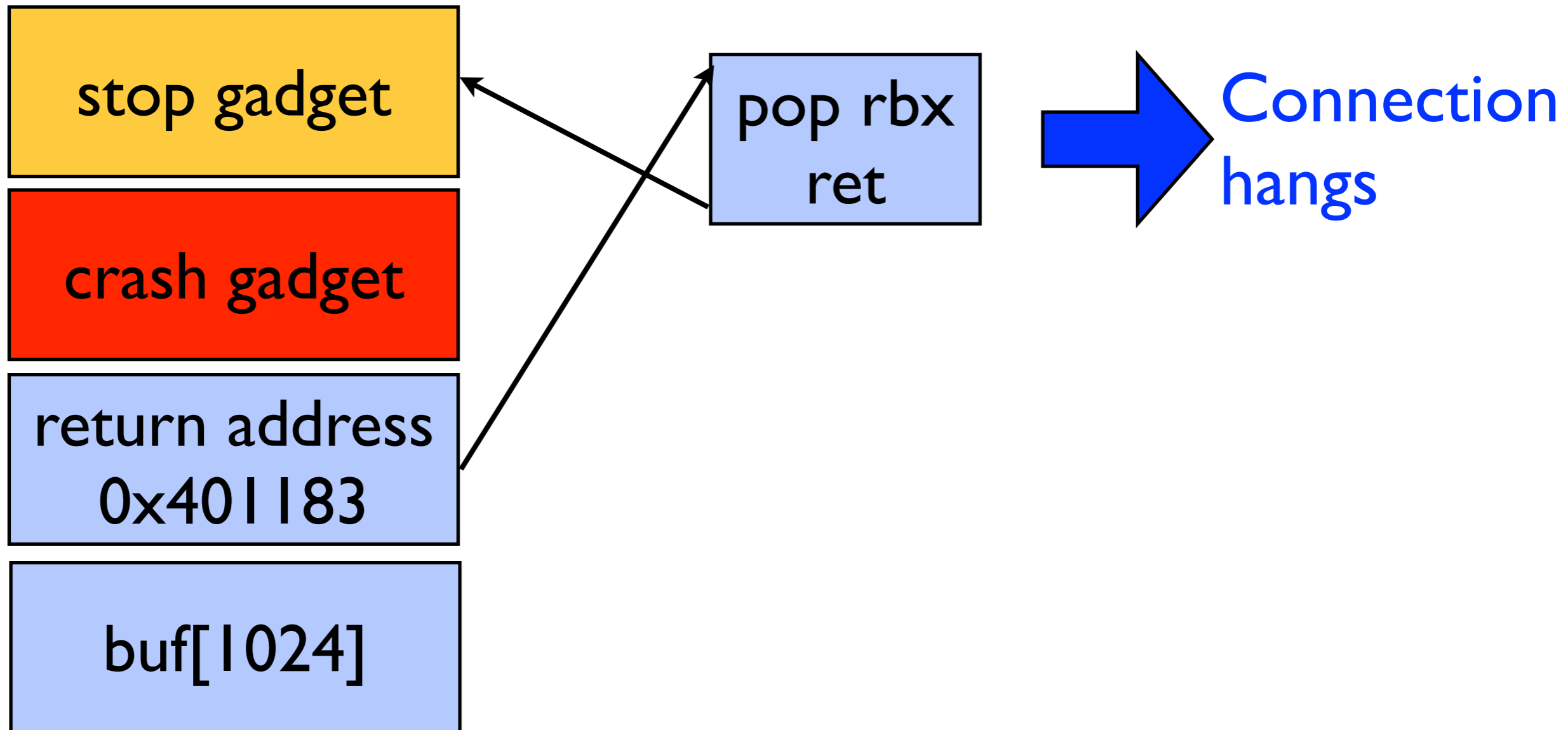
Finding the BROOP gadget

Stack:



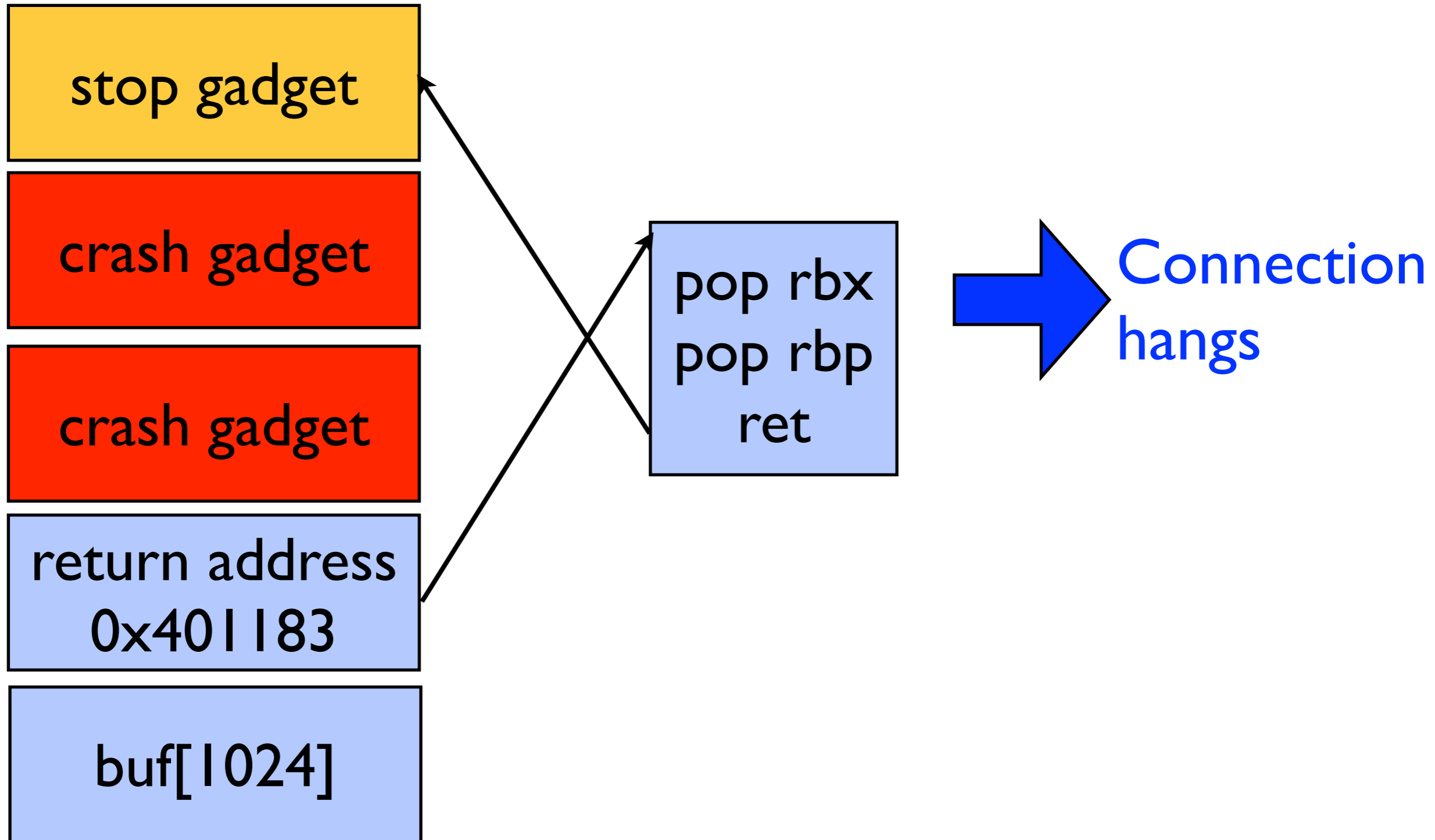
Finding the BROOP gadget

Stack:



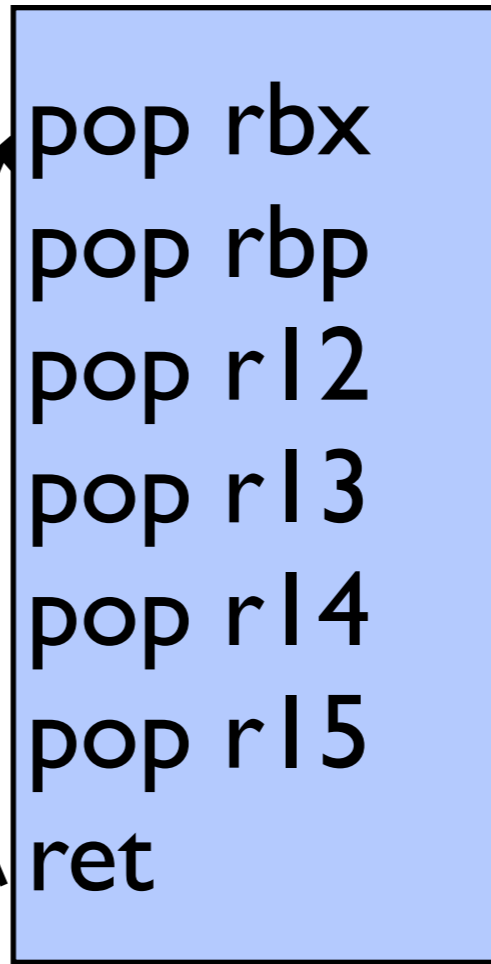
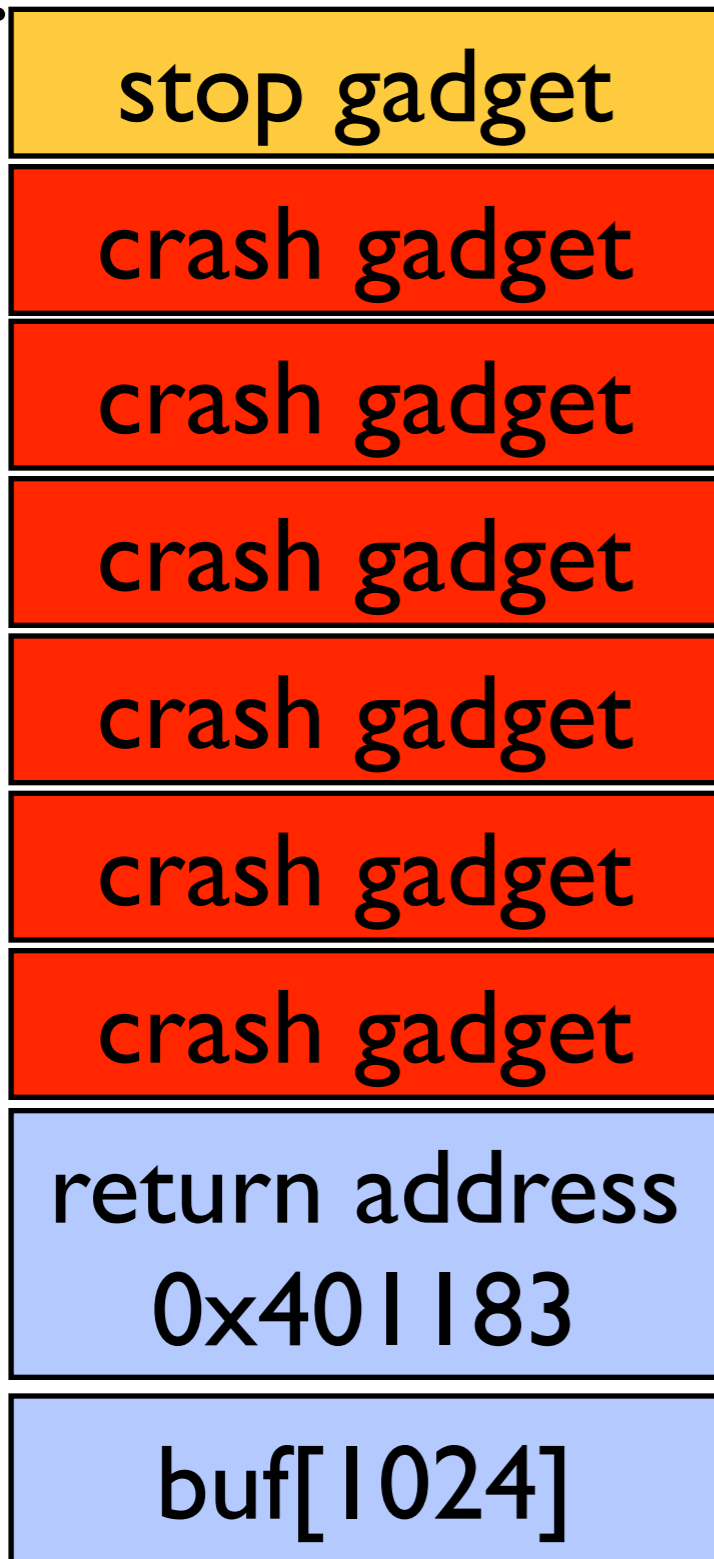
Finding the BROOP gadget

Stack:

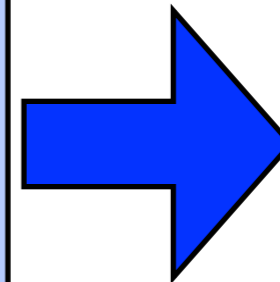


Finding the BRROP gadget

Stack:



BRROP gadget



Connection hangs

Attack so far

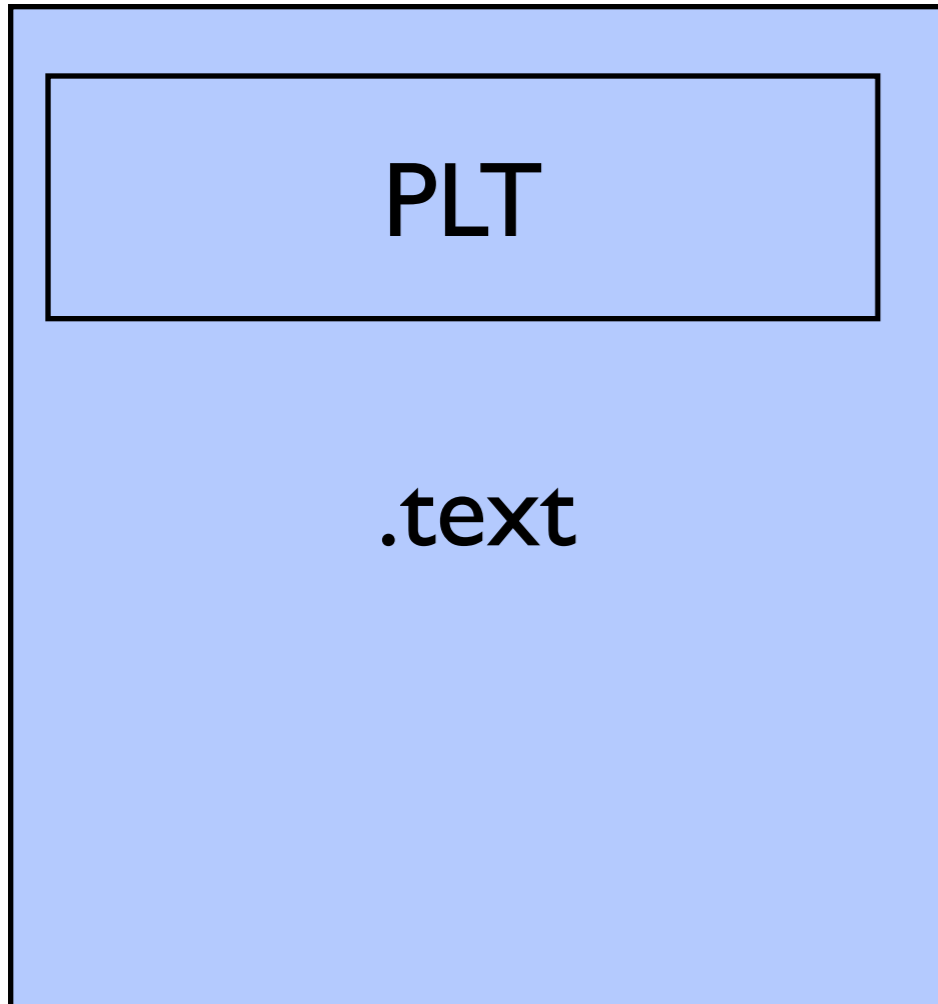
- Can control first two arguments to calls (using BROP gadget).
- Need to control third argument (using call strcmp).
- Need to find call write

Procedure Linking Table (PLT)



Procedure Linking Table (PLT)

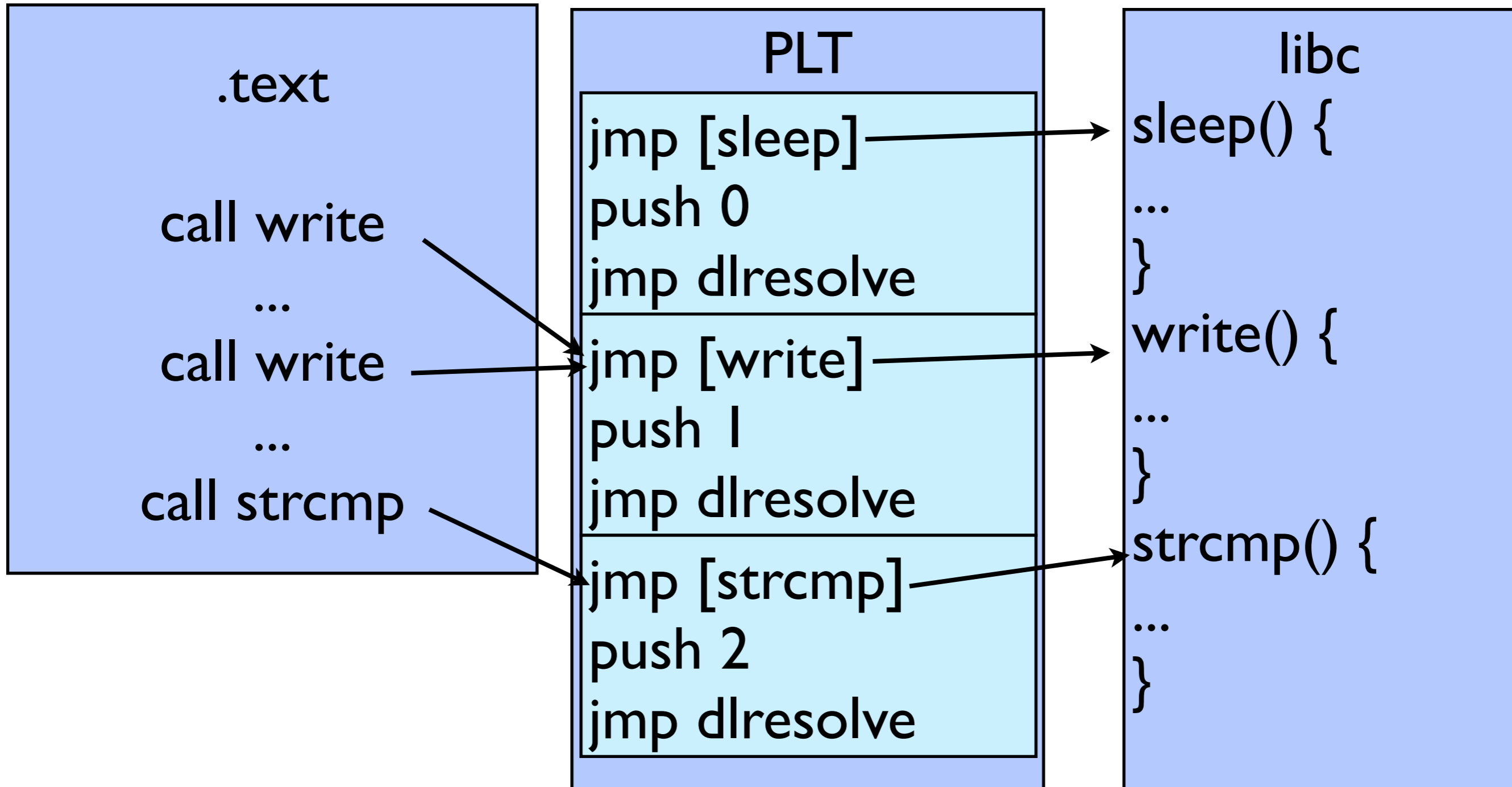
Origin: 0x400000



Procedure Linking Table (PLT)

Origin: 0x400000

[GOT dereference]



Finding the PLT

- Toward beginning of `.text`
- Each entry is 0x10 bytes apart, 0x10 aligned
- Most don't crash: syscalls - return EFAULT on bad args.
- Can check if entry + 6 succeeds too (dlresolve slow path)

Fingerprinting strcmp

arg1	arg2	result
readable	0x0	crash
0x0	readable	crash
readable	readable	nocrash

- Use return address (.text) as readable

Can now control three arguments:
strcmp sets RDX to length of string

Finding write

- `strcmp(origin, origin) // sets RDX / write length to 7 - ELF header.`
- set `rdi` to socket number, `rsi` to origin/addr
- try calling candidate PLT function.
- check if data received on socket.
- chain writes with different FD numbers to find socket. Use multiple connections.

Launching a shell

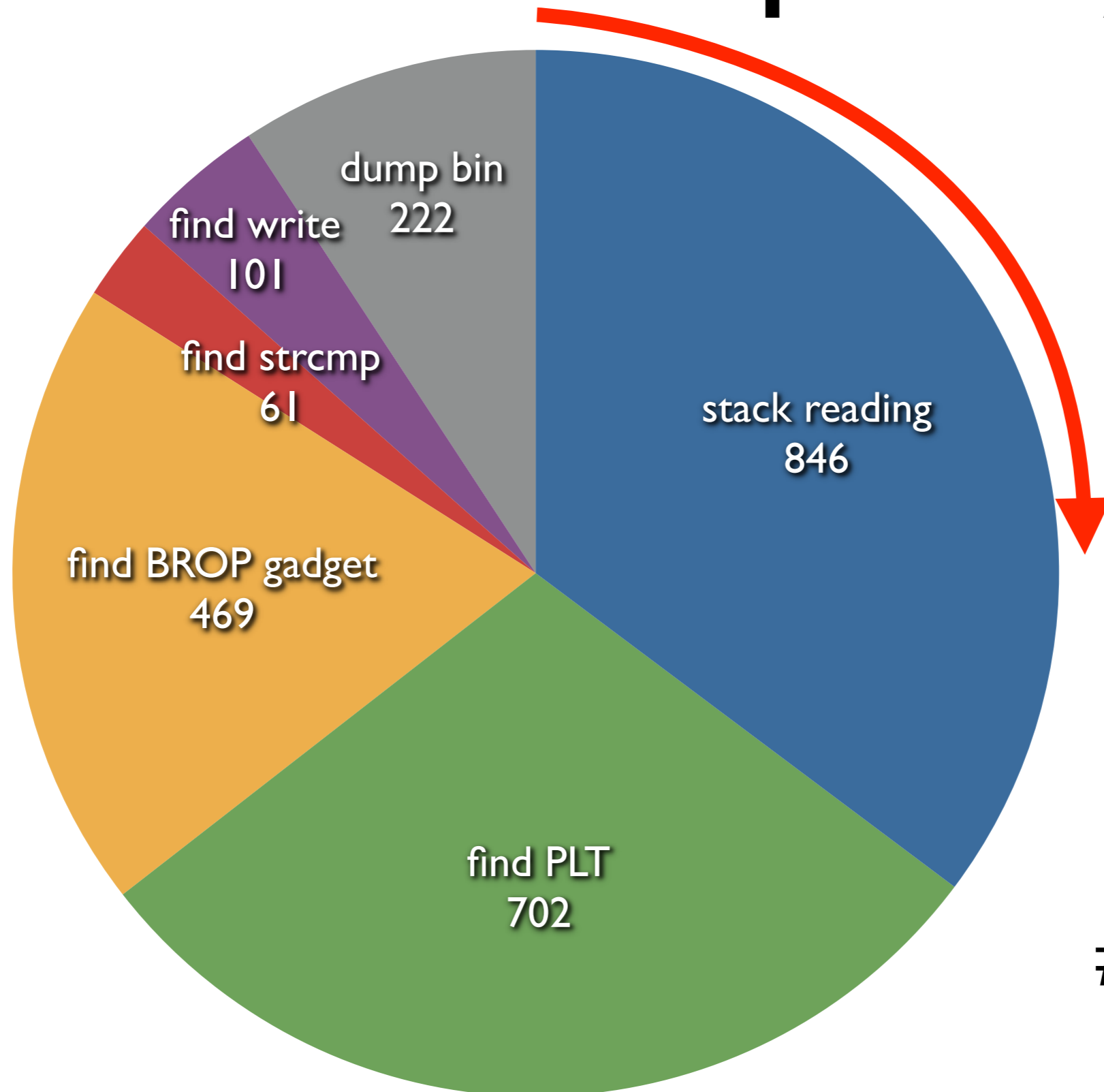
1. dump binary to network - not blind anymore!
2. dump symbol table to find PLT calls.
3. redirect stdin/out to socket:
 - `dup2(fd, 0);`
 - `close(0); fcntl(sock, F_DUPFD, 0);`
4. read() “/bin/sh” from socket to writable
 - writable can be environ (from symbol table)
5. `execve(“/bin/sh”, 0, 0)`

Braille

- Fully automated: from crash to shell.
- 2,000 lines of Ruby.
- Needs function that will trigger overflow:
 - nginx: 68 lines.
 - mysql: 121 lines.
 - proprietary service: 35 lines.

```
try_exp(data) → true  crash  
              false no crash
```

Attack complexity



of requests
for nginx

BROP conclusions

- It's scary what a motivated attacker can do: e.g., hack unknown binaries.
- Hackers can get past point solutions - need principled solutions:
 - OS: rerandomize ASLR & canaries per-fork
 - Compiler: Encrypt or MAC pointers?
 - Application: Dune (privilege separation)?