ROP and CFI

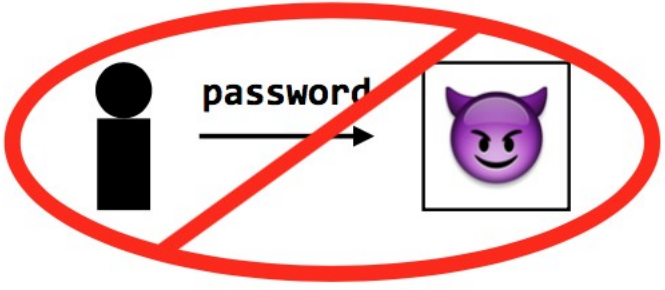
IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Key Problem of Password

Key problem: once you send a password to the server, it can impersonate you

- Solved part of the problem by hashing the password database
- But still sending the password to the server to verify on login

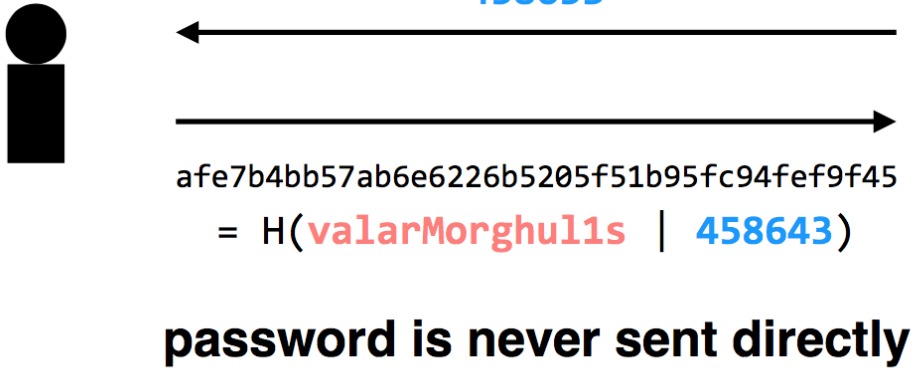


Tech 1: challenge-response scheme

Challenge-response

- Server chooses a random value **R**, sends it to client
- Client computes **H(R + password)** and sends to server
- Server checks if this matches its computation of hash with expected password
- If the server didn't already know password, still doesn't know

Tech 1: challenge-response scheme



password is never sent directly

valid server

username	password
arya	valarMorghul1s
jon	w1nterIsC0ming
sansa	LemonCakesForever
hodor	hodor

server computes
H(valarMorghul1s | 458643)
and checks

Adversary only learns H(valarMorghul1s | 458643); can not recover the password from that

Tech 2: use passwords to authenticate the server

Make the server prove it knows your password

- Client chooses **Q**, sends to server, server computes **H(Q + password)**, replies
- Only the authentic server would know your password!

Unfortunately, not many systems use this in practice

- In part because app developers just care about app authenticating user...

Tech 2: use passwords to authenticate the server

Complication: could be used with first scheme to fool server!

- First, try to log into the server: get the server's challenge **R**
- Then, decide you want to test the server: send it the same **R**
- Send server's reply back to it as response to the original challenge

Principle: be explicit, again!

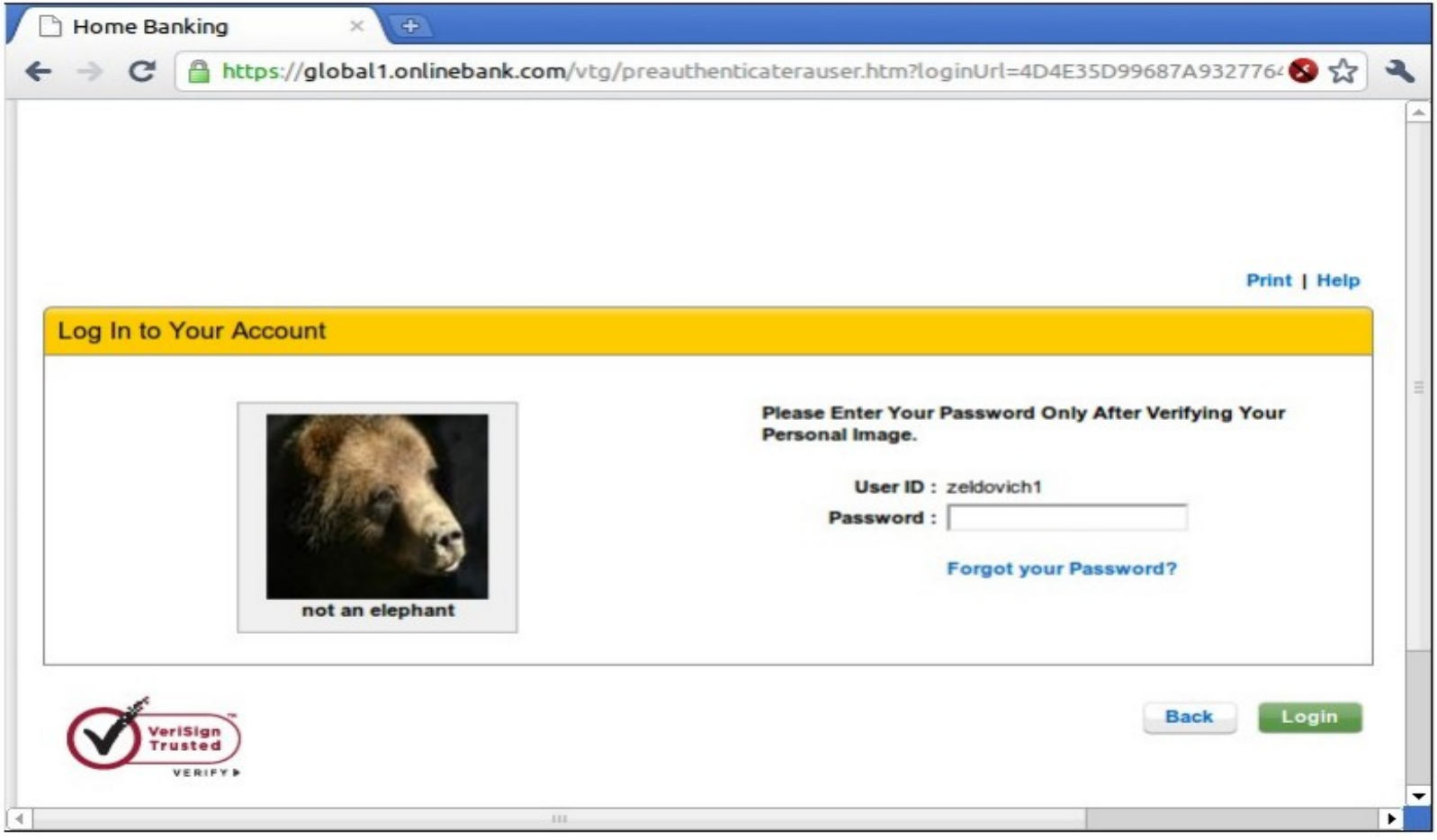
- E.g., hash the intended recipient of response (e.g., client or server), and have the recipient verify it

Tech 3: turn offline into online attack

Turn phishing attacks from offline into online attacks

- If adversary doesn't have the right image, users will know the site is fake
- Adversary could talk to real site, fetch image for each user that logs in

"Sitekey"


Home Banking
https://global1.onlinebank.com/vtg/preauthenticaterauser.htm?loginUrl=4D4E35D99687A932776
Print | Help
Log In to Your Account
not an elephant
Please Enter Your Password Only After Verifying Your Personal Image.
User ID : zeldovich1
Password :
Forgot your Password?
VeriSign Trusted
VERIFY
Back
Login

Tech 3: turn offline into online attack

Why is it still useful, then?

- Requires more efforts on adversary's part to mount attack
- Even if adversary does this, bank can detect it
- Watch for many requests coming from a single computer
- That computer might be trying to impersonate site
- Turns an offline/passive attack into an online/active attack

Key insight

- Don't need perfect security, small improvements can help

Tech 4: Specific password

Make passwords specific to a site

- Instead of sending password, send H(servername + password)
- Just like a basic password scheme, from the server's point of view

Except impersonator on another server gets diff passwd

Recommendation

- E.g., LastPass

Tech 5: one-time passwords

One-time Password

- If adversary intercepts password, can keep using it over and over
- Can implement one-time passwords: need to use a different password every time

Design: construct a long chain of hashes.

- Start with password and salt, as before
- Repeatedly apply hash, n times, to get n passwords
- Server stores x = H(H(H(H(...(H(salt+password)))))) = H^n(salt+password)

To authenticate, send token=$H^{\{n-1\}}$(salt+password)

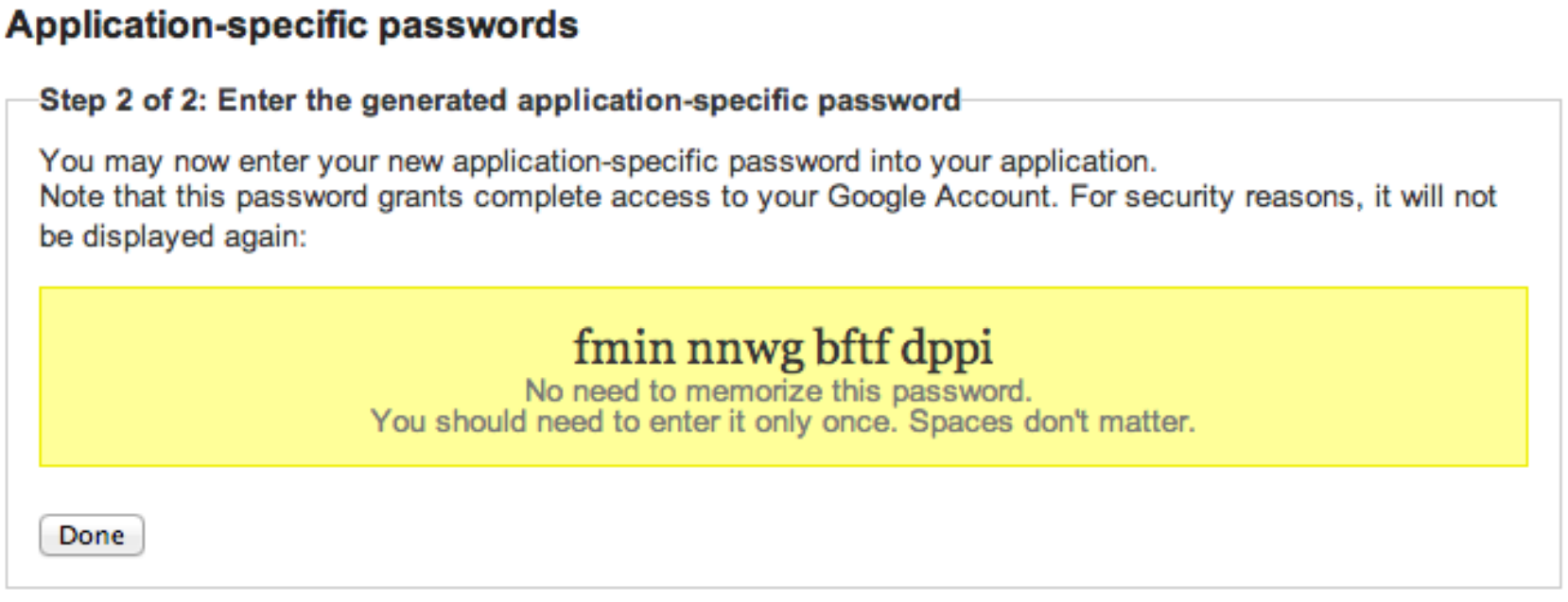
- Server verifies that x = H(token), then sets x <- token
- User carries a printout of a few hashes, or uses smartphone to compute them.

Tech 5: one-time passwords

Alternative design: include time in the hash (Google's 2-step verification)

- Server and user's smartphone share some secret string K
- To authenticate, smartphone computes H(K || current time)
- User sends hash value to server, server can check a few recent time values.

Google's App-specific password



Reeder on MBA	Dec 2, 2012	Dec 19, 2012	[Revoke]
Chrome on MBA	Dec 2, 2012	Dec 2, 2012	[Revoke]
Gmail on iPhone	Dec 10, 2012	Unavailable	[Revoke]
Test for CSE	Dec 20, 2012	Unavailable	[Revoke]

Tech 6: bind authentication and request authorization

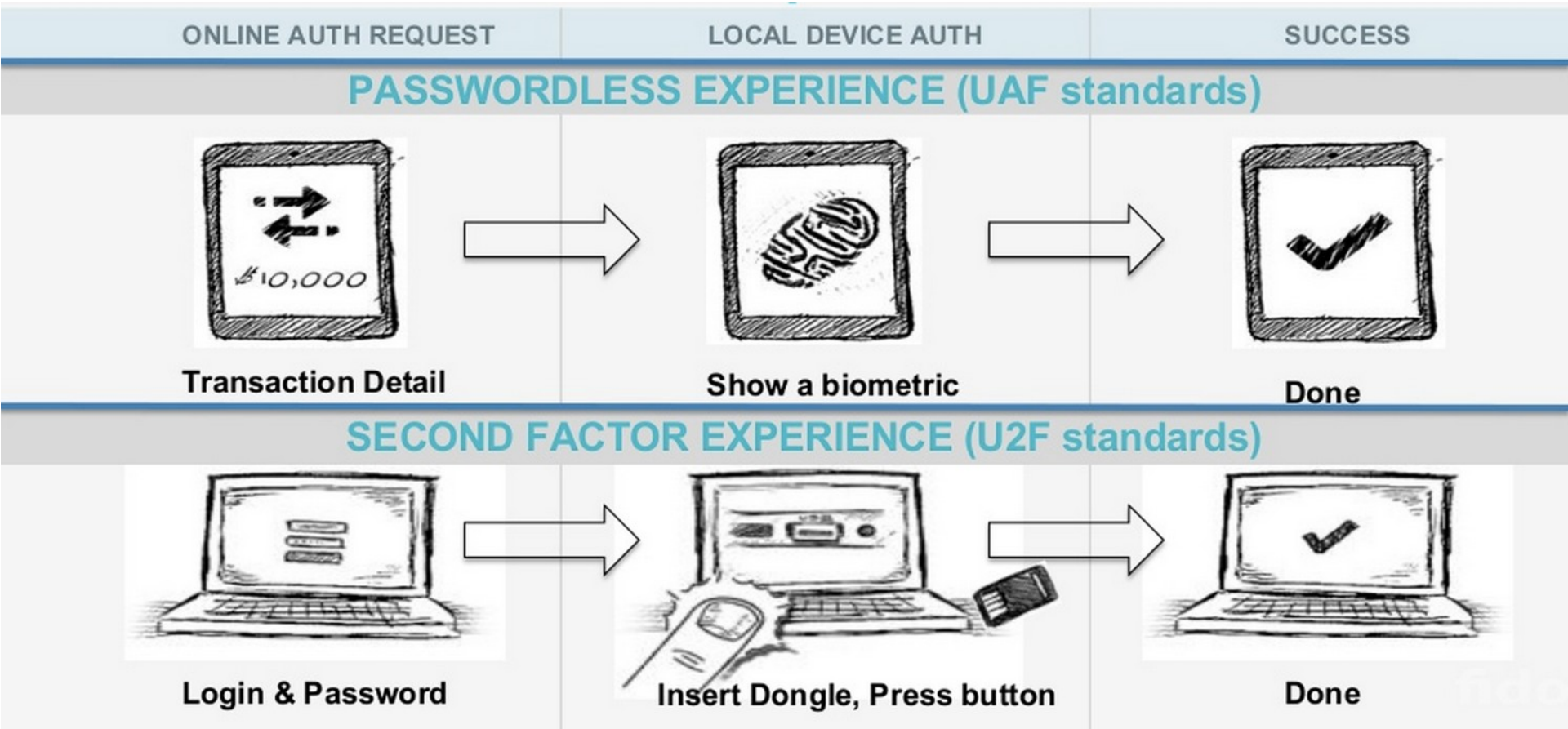
One way to look at problem: sending password authorizes any request

- Even requests by adversary that intercepts our password
- A different design: use password to authenticate any request
- **req = { username, "write XX to exam.txt", H(password + "write ..") }**

Server can check if this is a legitimate req from user using password

- Even if adversary intercepts request, cannot steal/misuse password
- In practice, don't want to use password, use some session token instead
- Could combine well with one-time passwords

Tech 7: FIDO: Replace the Password



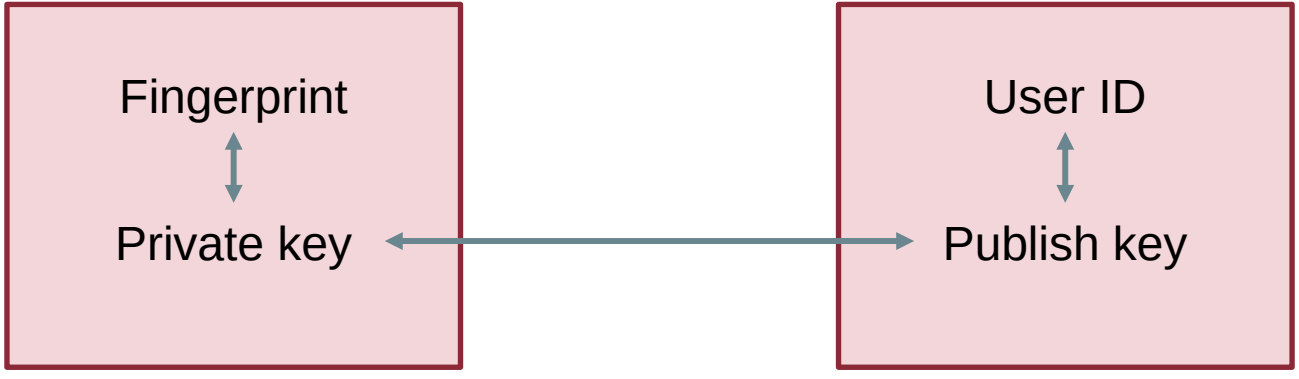
PRESS RELEASE

May 5, 2022

Apple, Google, and Microsoft commit to expanded support for FIDO standard to accelerate availability of passwordless sign-ins

Faster, easier, and more secure sign-ins will be available to consumers across leading devices and platforms

Three Bindings



Bootstrapping

How to initially set a password for an account?

- If adversary can subvert this process, cannot rely on much else
- MIT: admissions office vets each student, hands out account codes
- Many web sites: anyone with an email can create a new account

Changing password (e.g., after compromise)

- Need some additional mechanism to differentiate user vs. attacker
- MIT: walk over to accounts office, show ID, admin can reset password
- Many sites: additional "security" questions used to reset password

Password bootstrap / reset mechanisms are part of the security system

- Can sometimes be weaker than the password mechanisms
- Sarah Palin's Yahoo account was compromised by guessing security Q's
- Personal information can be easy to find online

SECURITY PRINCIPLES

Principle of Least Privilege

Deal with bugs / incomplete mediation

- Any component that can arbitrarily access a resource must invoke the guard
- If component has a bug or design mistake, can lead to incomplete mediation
- General plan: reduce the number of components that must invoke the guard
  - E.g., arrange for DB server to check permissions on records returned, then security does not depend as much on *lookup.cgi*

Least Trust

Privileged components are "trusted"

- **Trusted is “bad”**: you are in trouble if a trusted component breaks
- **Untrusted components are “good”**: does not matter if they break
- Good design has **few trusted components**, other things do not affect security

Users Make Mistakes

Social engineering, phishing attacks

- How many of you really read the permission acquired by apps during installing?

Good idea: threat model should not assume users are perfect

Cost of Security

Security VS. Availability

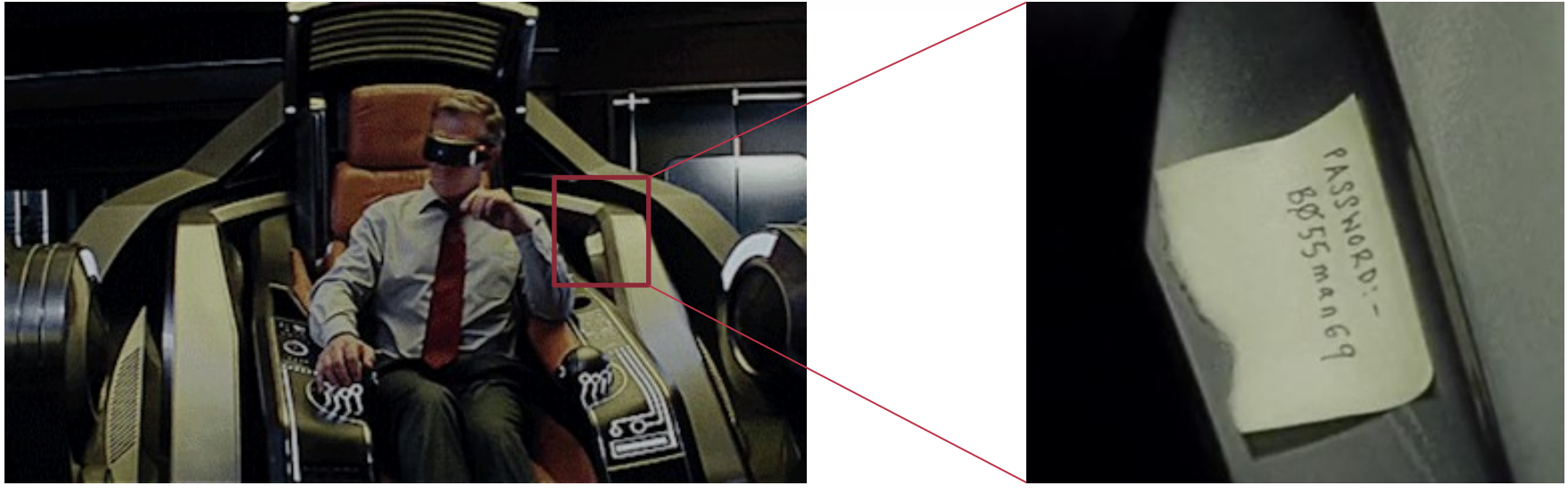
- E.g., system requires frequent password changes -> users may write them down

How far should a teacher go to protect the exam file?

- Put the file on a separate computer, to avoid sharing a file system?
- Disconnect computer from the network, to avoid remotely exploitable OS bugs?
- Put the server into a separate machine room?
- Get a guard to physically protect the machine room?
- ...

Good idea: cost of security mechanism should be commensurate with value

B055man69



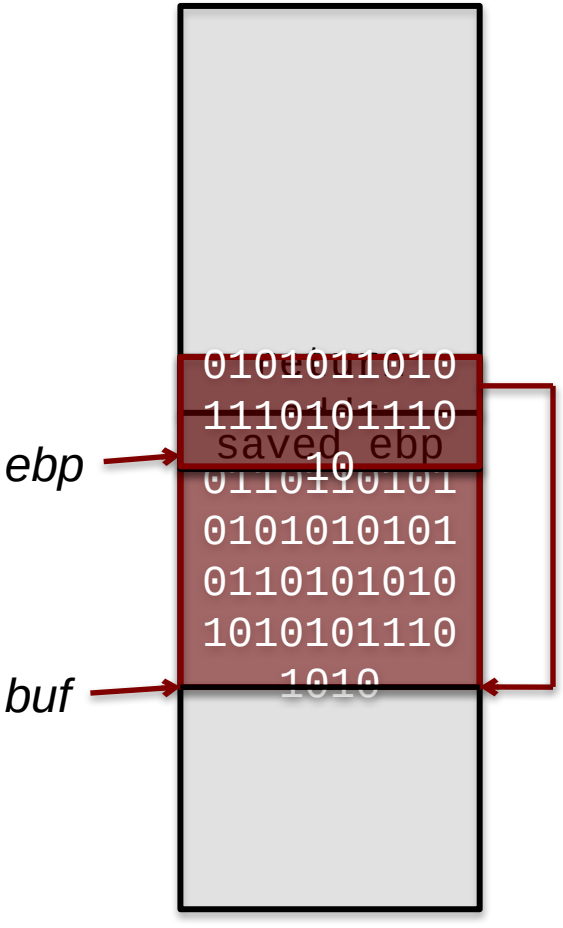

return-Oriented
PROgRammIng

Review: Stack Buffer Overflow

A Typical Buffer Overflow Attack

- Inject malicious code in buffer
  - E.g., shellcode
- Overwrite return address to buffer
- Once return, the malicious code runs

```
void function(char *str) {
   char buf[16];
   strcpy(buf,str);
}
```



Shellcode Sample to Open "/bin/sh" (21 bytes)

```
int main(){

  char sc[] =  "\x6a\x0b"                  // push byte 0xb
               "\x58"                      // pop eax (now eax=0xb)
               "\x99"                      // cdq
               "\x52"                      // push edx (now edx=0)
               "\x68\x2f\x2f\x73\x68"        // push dword 0x68732f2f
               "\x68\x2f\x62\x69\x6e"        // push dword 0x6e69622f
               "\x89\xe3"                  // mov ebx, esp
               "\x31\xc9"                  // xor ecx, ecx (now ecx=0)
               "\xcd\x80";                 // int 0x80

  ((void (*)()) sc)();
}
```

More can be found on https://www.exploit-db.com

Name	Registers						Definition
	eax	ebx	ecx	edx	esi	edi	
sys_execve	0x0b	char __user *	char __user *	char __user *	**struct pt_regs** *	-	**arch/alpha/kernel/entry.S:925**

Defense: DEP (Data Execution Prevention)

Execute *Code*, not *Data*

Data areas marked non-executable

- Stack marked non-executable
- Heap marked non-executable

Enforced by hardware

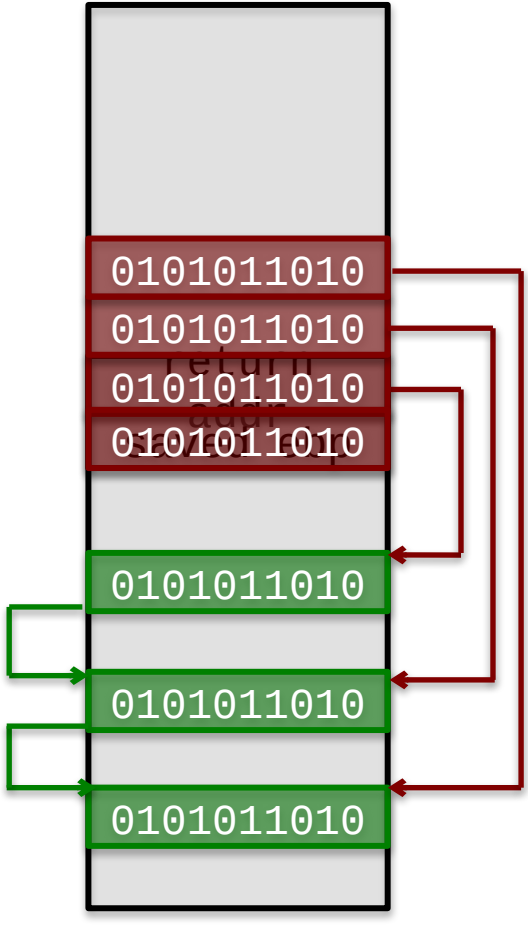
- NX: Non-eXecutable (the name on hardware)

You can load your *shellcode* in the stack or the heap…but you can't jump to it to execute

Attack: Code Reuse (instead of inject) Attack

ROP: Return-oriented Programming

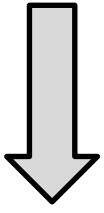
- Find code gadgets in existed code base
  - Usually 1-3 instructions, ends with 'ret'
  - In libc and application, intended and unintended
- Push address of gadgets on the stack
- Leverage 'ret' at the end of gadget to connect each code gadgets
- **No code injection**



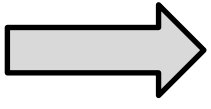
Code Execution – The ROP Way

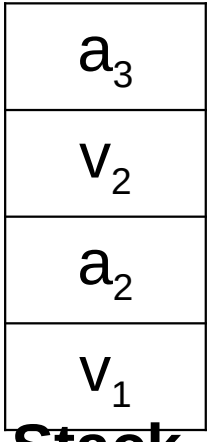
```
Mem[v2] = v1
```

Desired Logic



```
mov %eax v1;
mov %ebx v2;
mov [%ebx], %eax
```





Stack

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

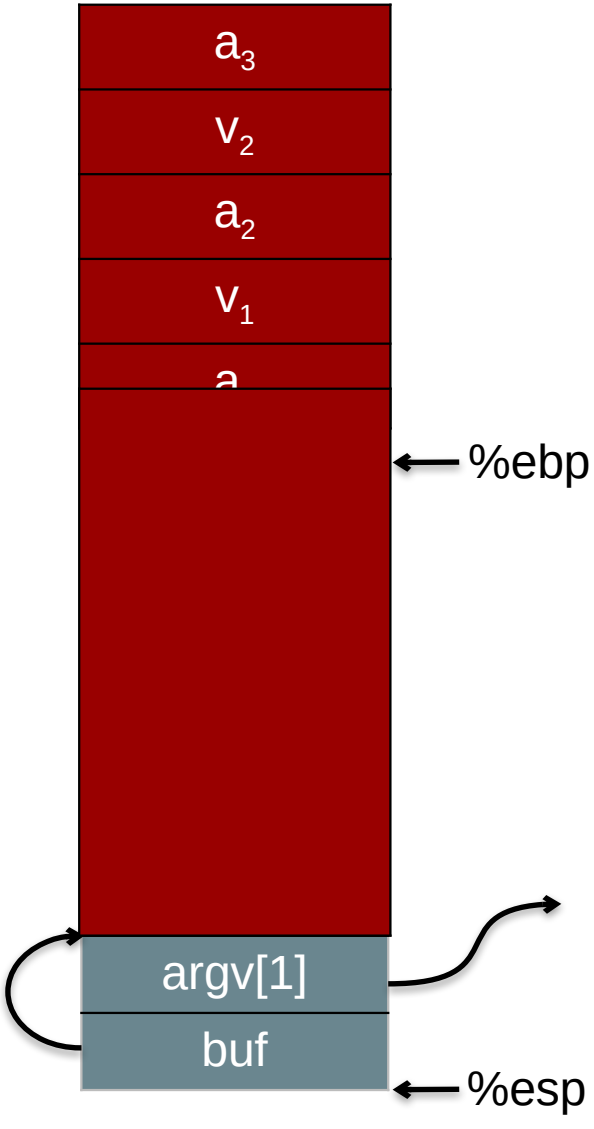
Code Execution – The ROP Way

```
Mem[v2] = v1
```

Desired *Shellcode*

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

Desired store executed!




Return-Oriented Programming
is a lot like a ransom note, but instead of cutting out letters from magazines, you are cutting out instructions from .text segments

Defense

Hide the binary file

- No way to get any gadget

ASLR to randomize the code position

- Short for "Address Space Layout Randomization"
- Harder to find gadgets

Canary to protect the stack

- Try to detect stack overflow (e.g., overflow return address)

ASLR

Change the layout of memory space

Every time when a process is created

- Question: how about create by fork?

User-level as well as kernel-level

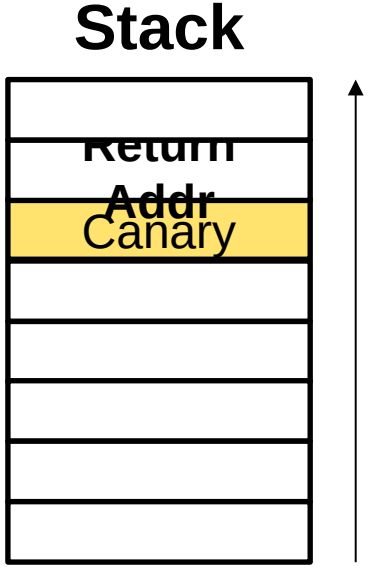
Canary

Embed "canaries" in stack frames and verify their integrity prior to function return

- Canary is just a random number
- Check canary before return, alert if not equal

StackGuard implemented as a GCC patch

- Program must be recompiled
- Performance overhead: 8% for Apache



CFI: CONTROL FLOW INTEGRITY

CFI: Control-Flow Integrity

Main idea: pre-determine control flow graph (CFG) of an application

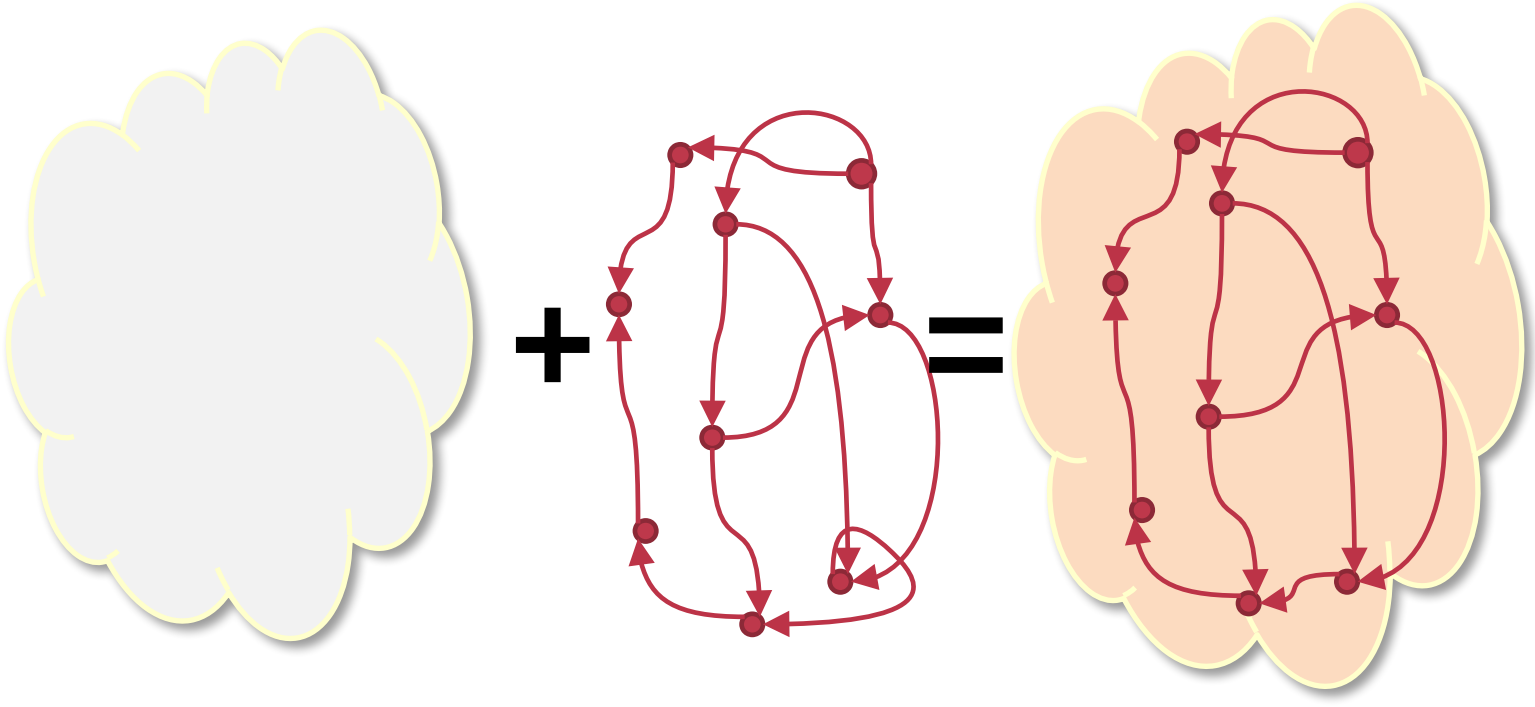
- Static analysis of source code
- Static binary analysis ← **CFI**
- Execution profiling
- Explicit specification of security policy

Execution must follow the pre-determined control flow graph

CFI Motivation

Anatomy of many software attacks

CFI Idea



Branch Types

- Direct Branches
  - Direct call
  - Direct jump
- Indirect Branches
  - Return
  - Indirect call
  - Indirect jump

In Apache and its libraries

Types	In Binary	Run-time
Direct call	16.8%	14.5%
Direct jump	74.3%	0.8%
Return	6.3%	16.3%
Indirect call	2.1%	0.2%
Indirect jump	0.5%	68.3%

- has 1 target: 94.7%
- <= 2 targets: 99.3%
- >10 targets: 0.1%

CFI: Binary Instrumentation

Use *binary rewriting* to instrument code with runtime checks

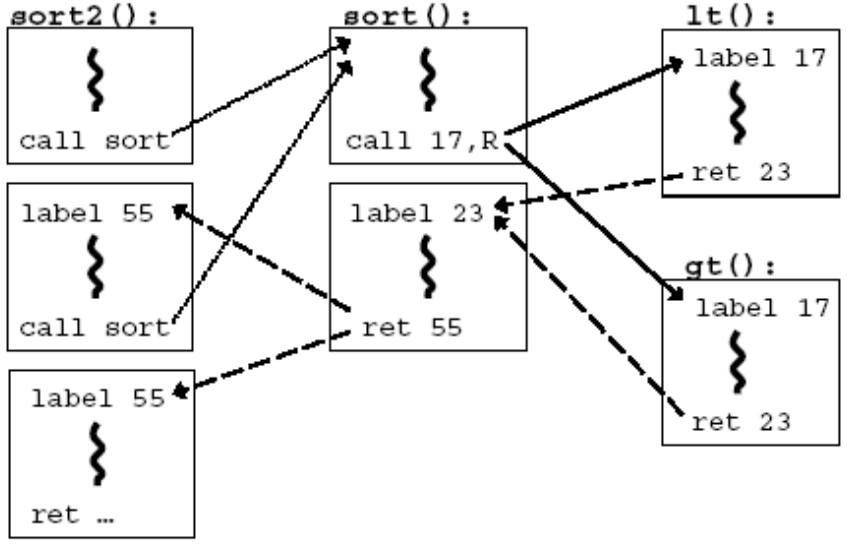
Inserted checks ensure that the execution always stays within the statically determined CFG

Whenever an instruction transfers control, destination must be valid according to the CFG

Goal: Secure even if the attacker has complete control over the thread's address space

CFG Example

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



CFI: Control Flow Enforcement

For each control transfer, determine statically its possible destination(s)

Insert a unique bit pattern at every destination

- Two destinations are equivalent if CFG contains edges to each from the same source
  - This is imprecise (why?)
- Use same bit pattern for equivalent destinations

CFI: Control Flow Enforcement

Insert binary code that at runtime will check whether the bit pattern of the target instruction matches the pattern of possible destinations

CFI: Example of Instrumentation

Original

```
jmp   ecx
```

Patched

```
cmp   [ecx], 12345678h
jne   error_label
lea   ecx, [ecx+4]
jmp   ecx
```

```
mov   eax, [esp+4]     ; dst
```

```
; data 12345678h       ; ID
mov   eax, [esp+4]     ; dst
```

CFI: Example of Instrumentation

```
mov  eax, 12345677h    ; load ID-1          3E 0F 18 05    prefetchnta          ; label
inc  eax               ; add 1 for ID       78 56 34 12       [12345678h]       ;    ID
cmp  [ecx+4], eax      ; compare w/dst      8B 44 24 04    mov  eax, [esp+4]    ; dst
jne  error_label       ; if != fail         ...
jmp  ecx               ; jump to label
```

Prefetchnta: prefetch memory to cache. Become a *nop* if not available.

Improving CFI Precision

Suppose a call from A goes to C, and a call from B goes to either C, or D
(when can this happen?)

- CFI will use the same tag for C and D, but this allows an "invalid" call from A to D
- Possible solution: duplicate code or inline
- Possible solution: multiple tags

Improving CFI Precision

Function F is called first from A, then from B; what's a valid destination for its return?

- CFI will use the same tag for both call sites, but this allows F to return to B after being called from A
- Solution: shadow call stack
  - Maintain another stack, just for return address
  - Intel CET to the rescue (not available yet)

CFI: Security Guarantees

Effective against attacks based on illegitimate control-flow transfer

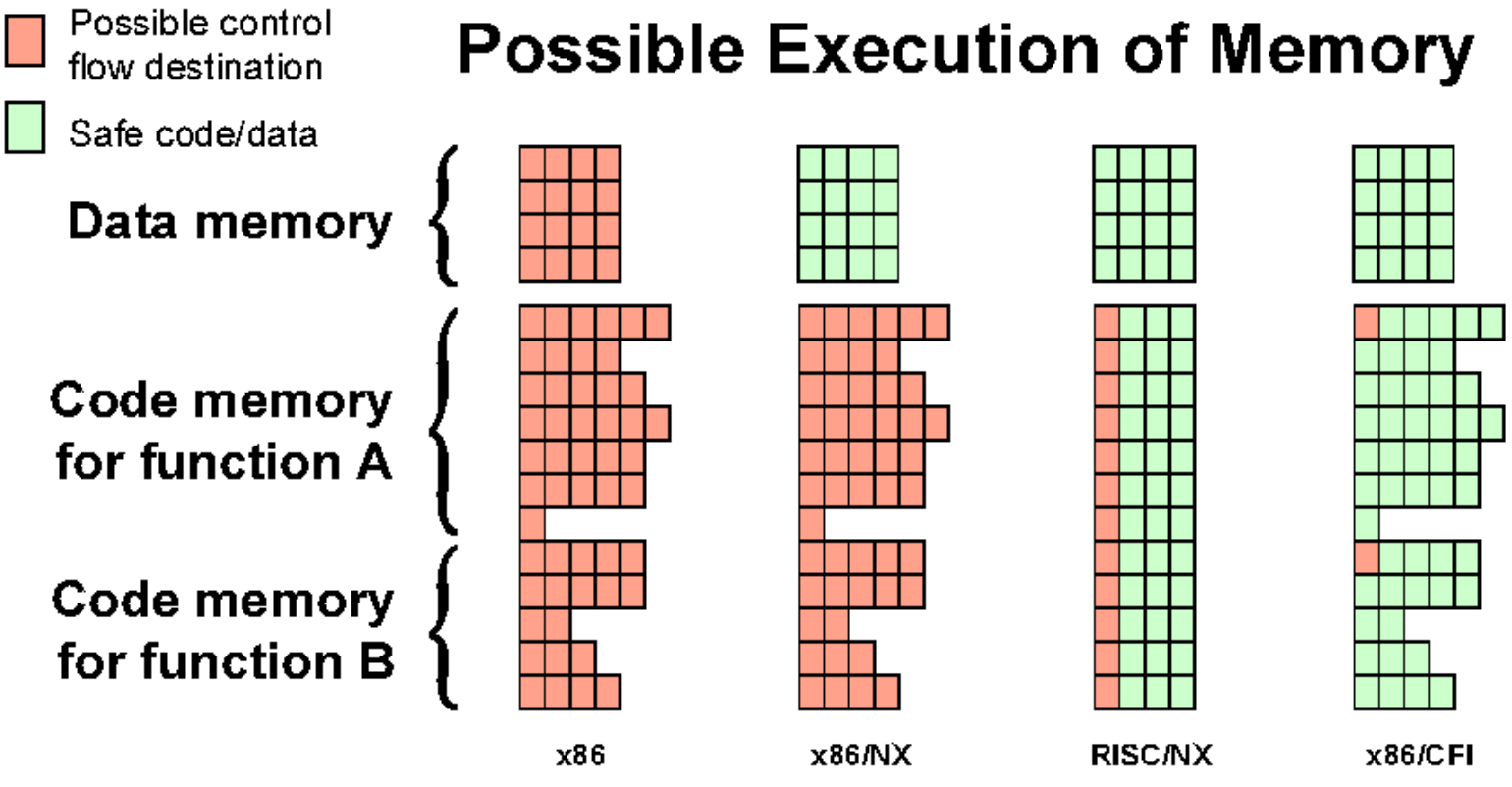
- Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge

Does not protect against attacks that do not violate the program's original CFG

- Incorrect arguments to system calls
- Substitution of file names
- Other data-only attacks

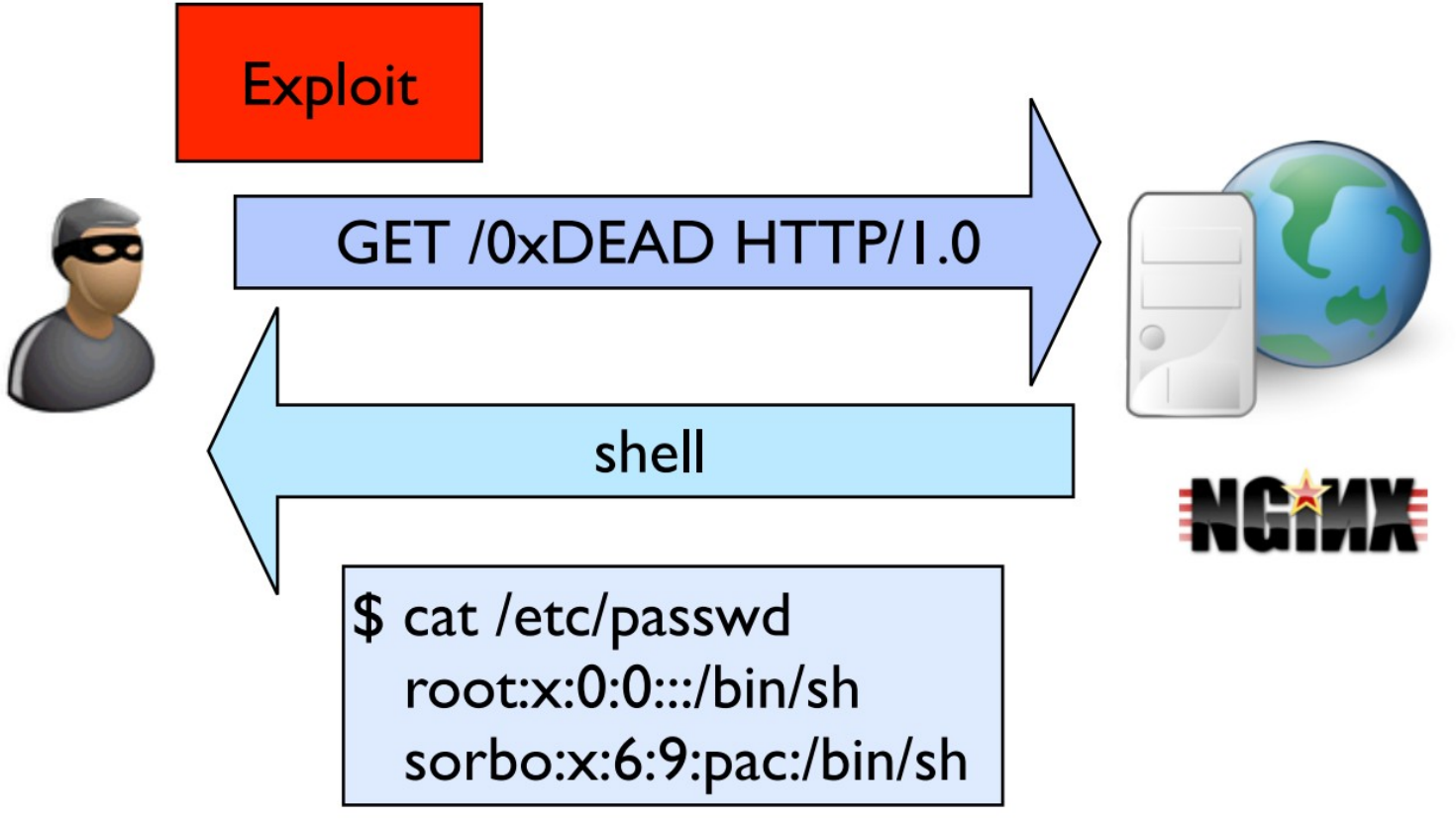
Possible Execution of Memory

[Erlingsson]

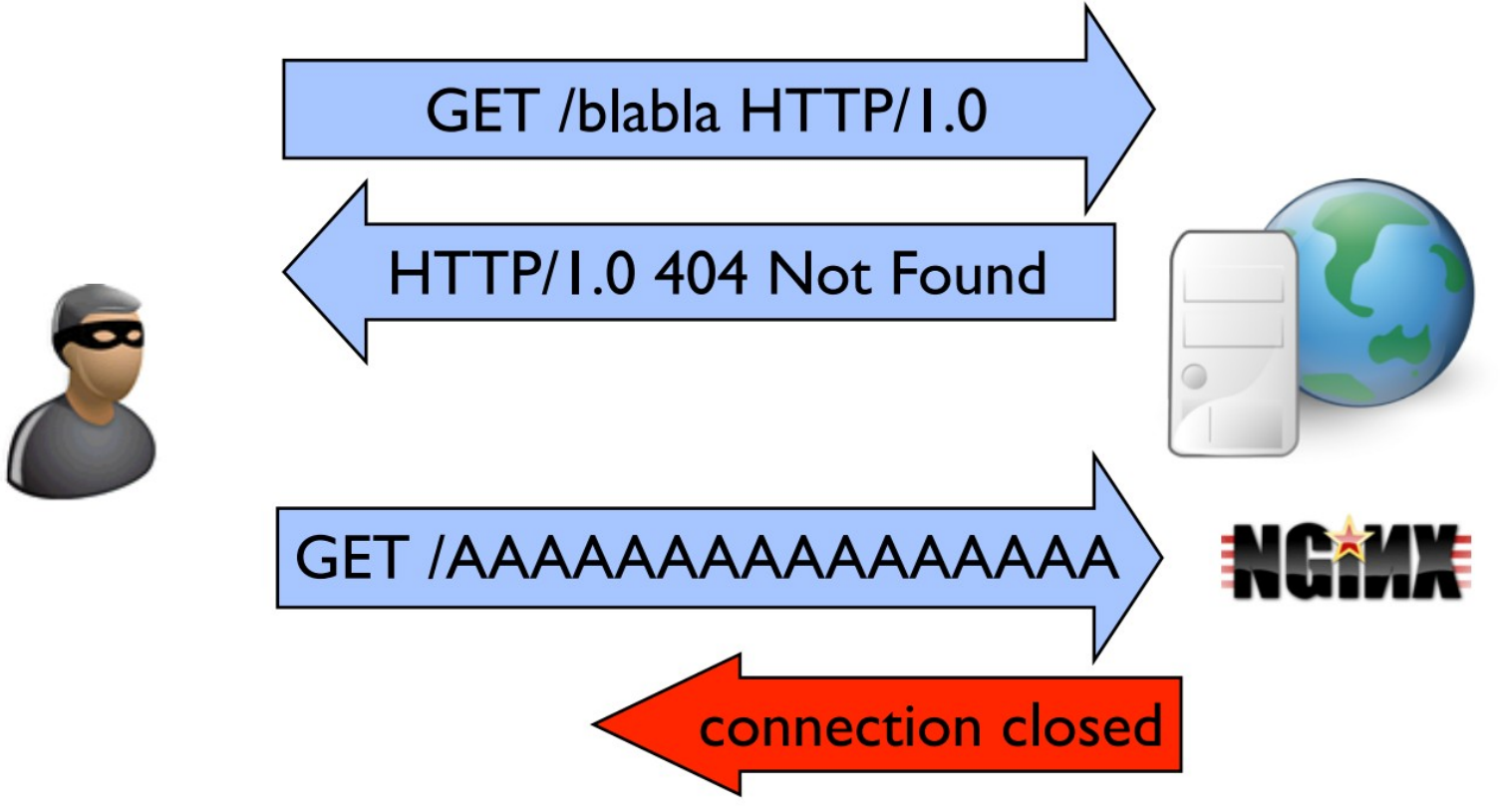


BLIND ROP

Hacking buffer overflows



Crash or no Crash?
Enough to build exploit



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

Exploit scenarios:

1. Open source

2. Open binary

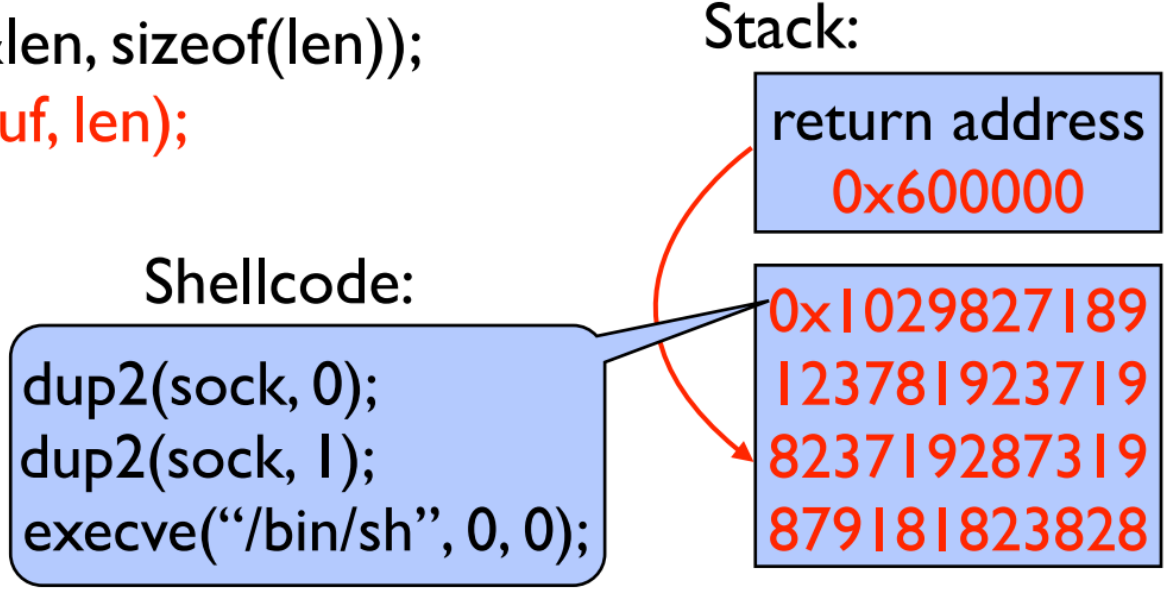
3. Closed-binary (and source)

Attack requirements

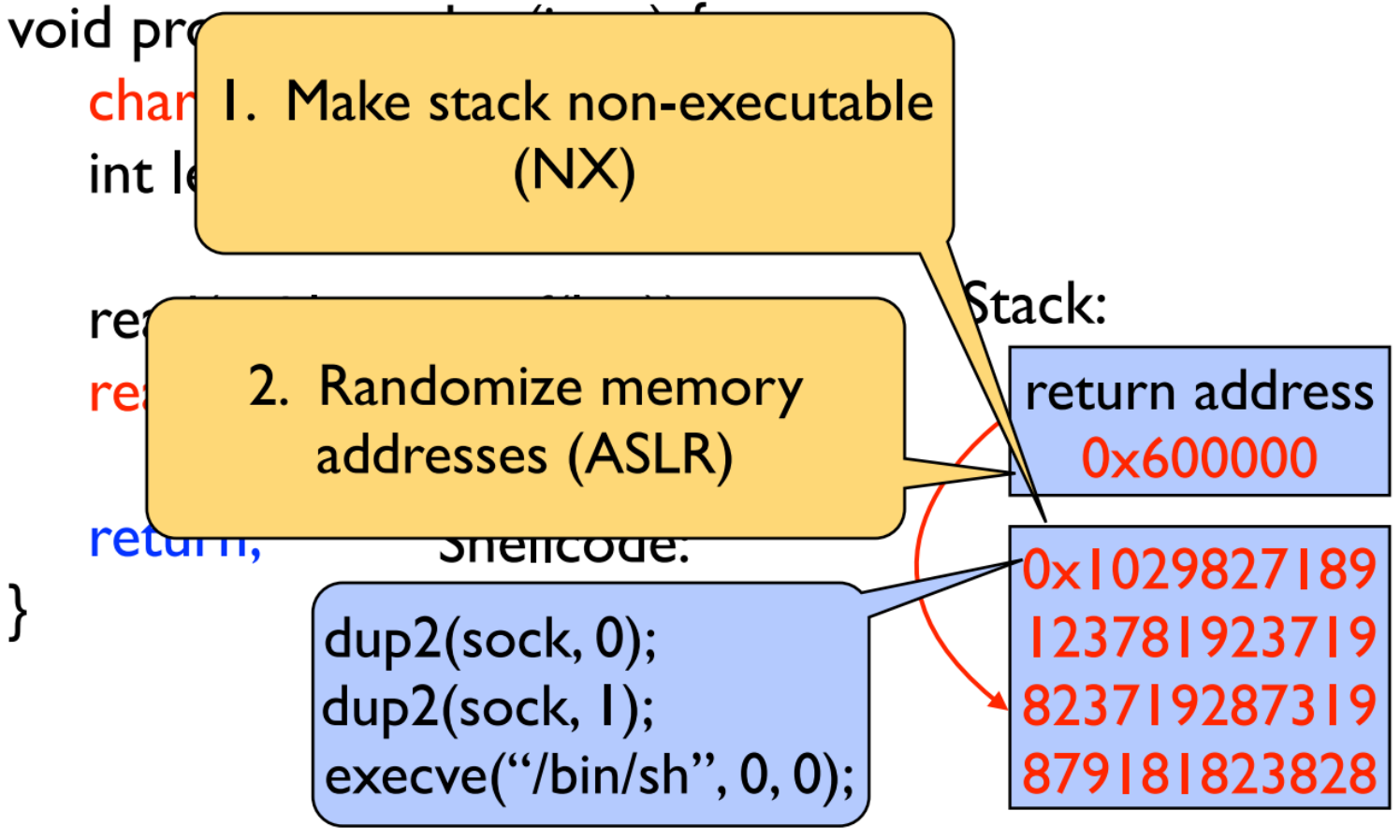
1. Stack vulnerability, and knowledge of how to trigger it.
2. Server process that respawns after crash
   - E.g., nginx, MySQL, Apache, OpenSSH, Samba.

Stack vulnerabilities

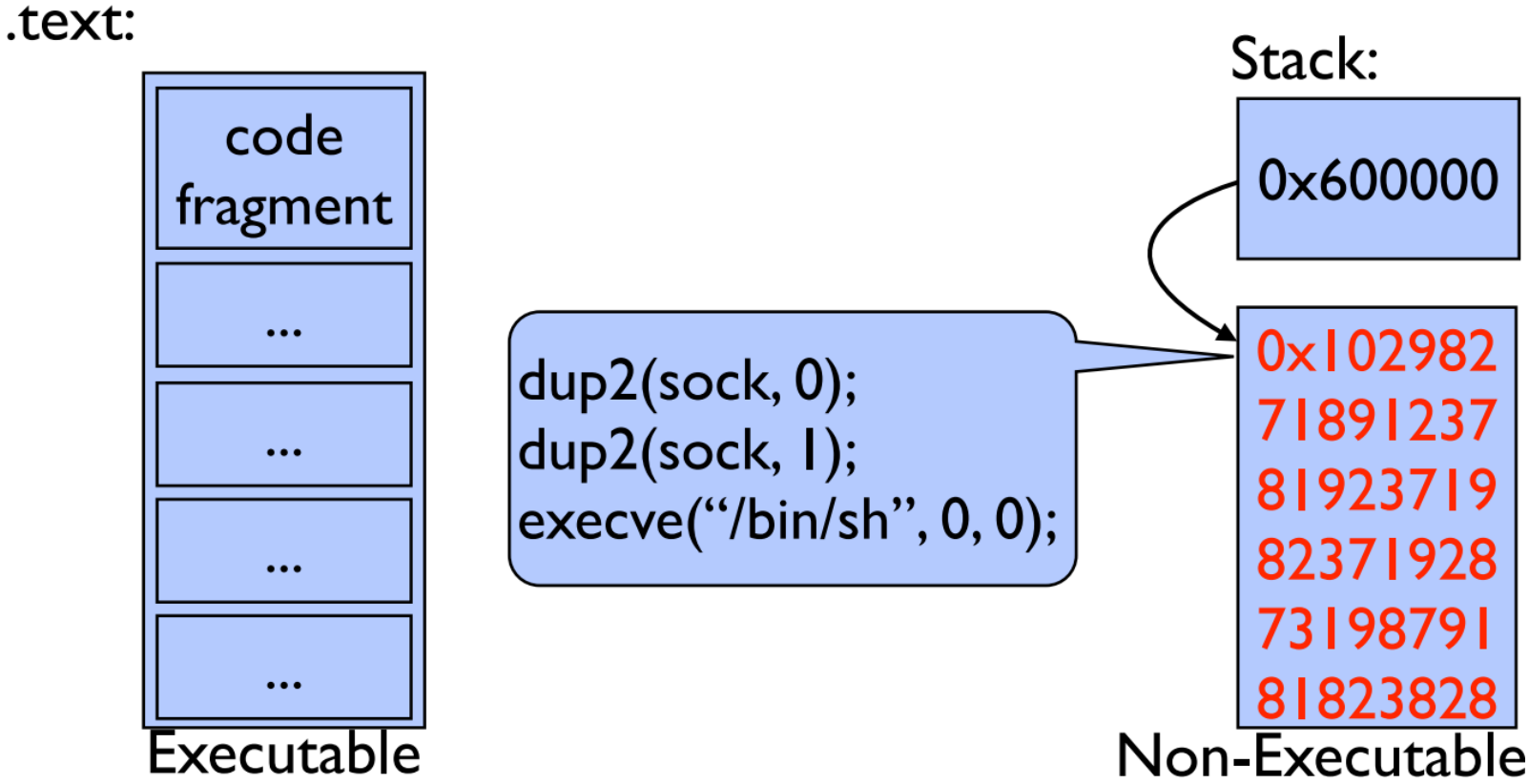
```
void process_packet(int s) {
    char buf[1024];
    int len;

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

    return;
}
```



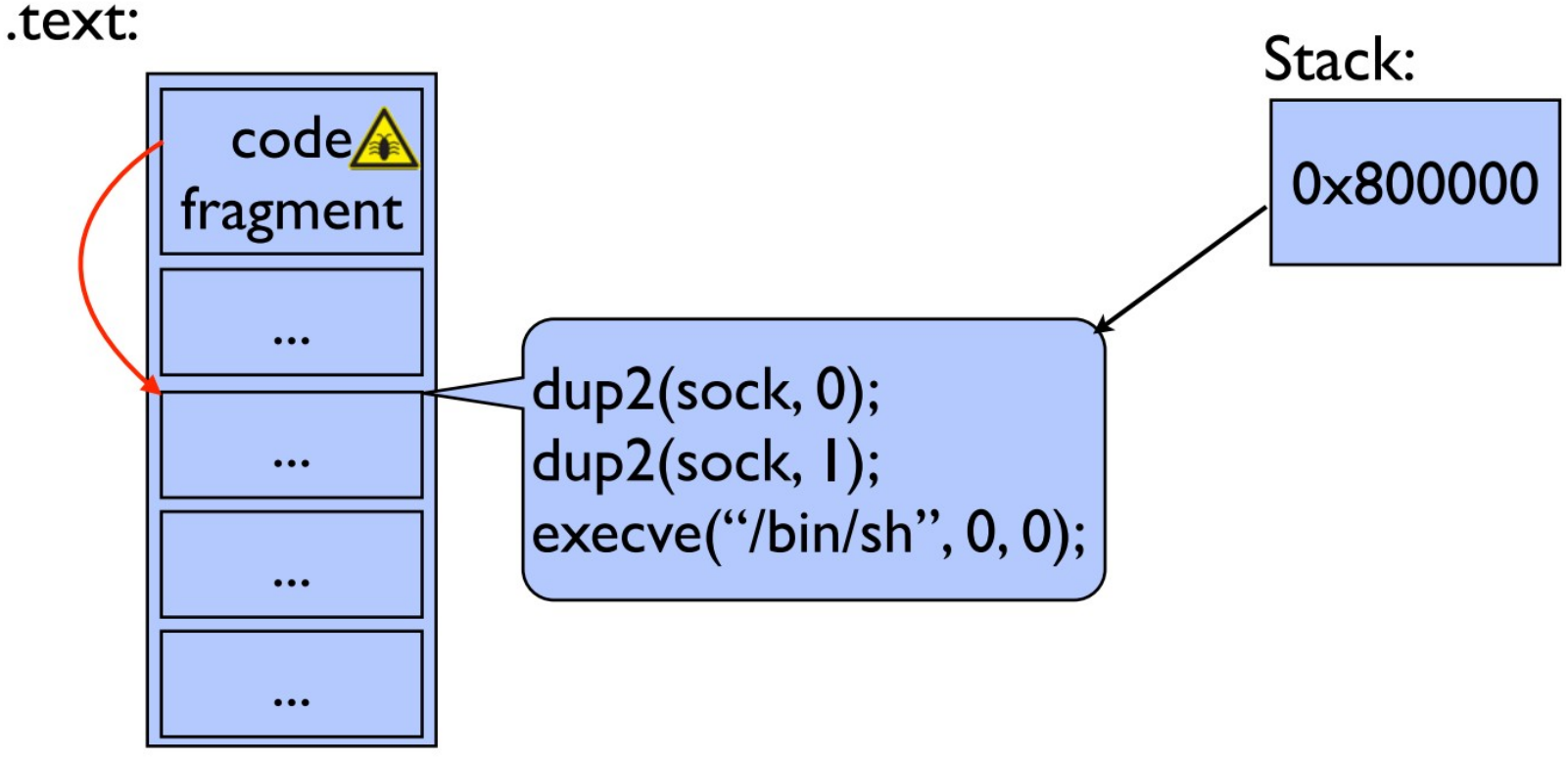
Exploit protections



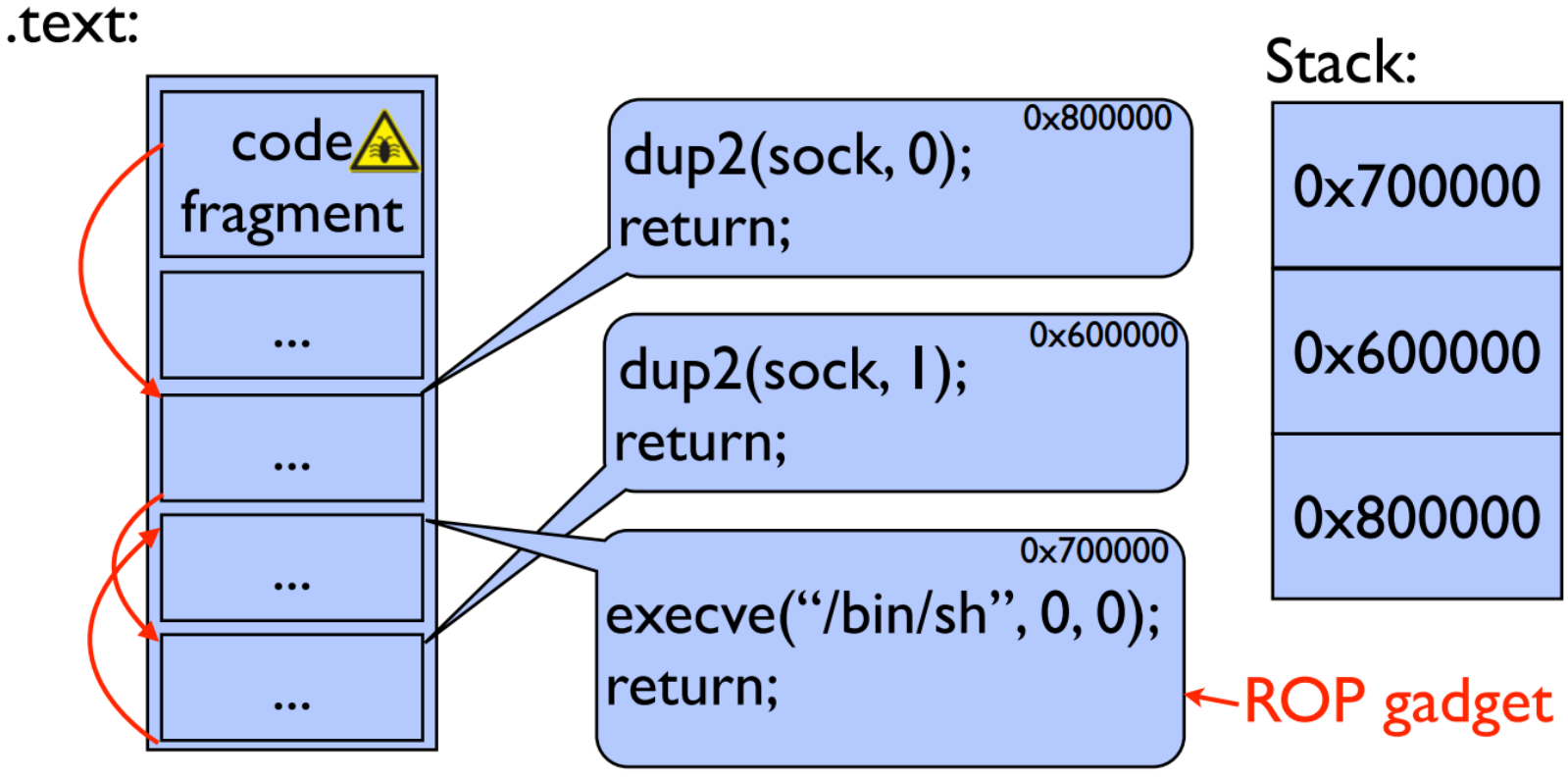
Return-Oriented Programming (ROP)



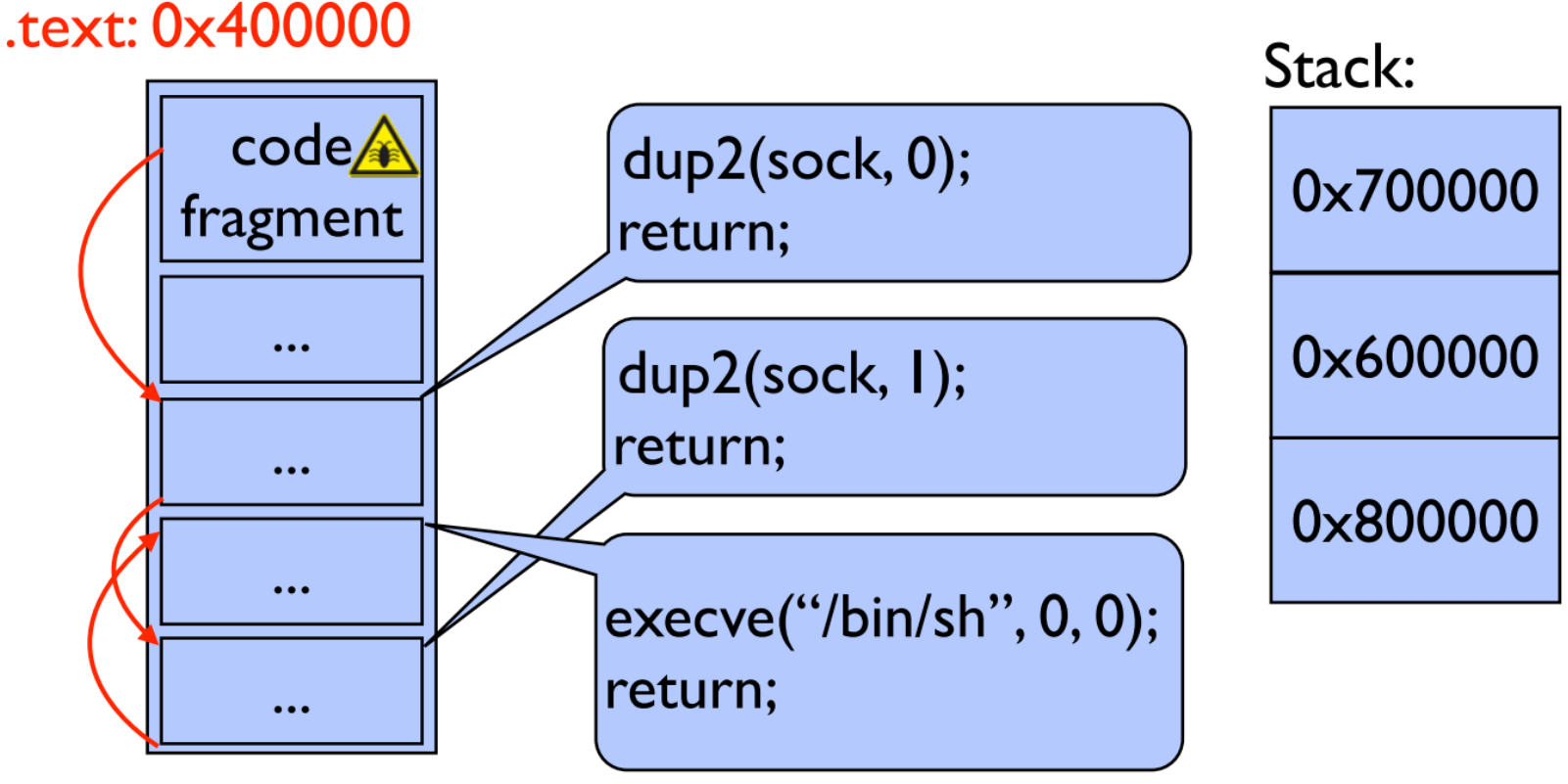
Return-Oriented Programming (ROP)



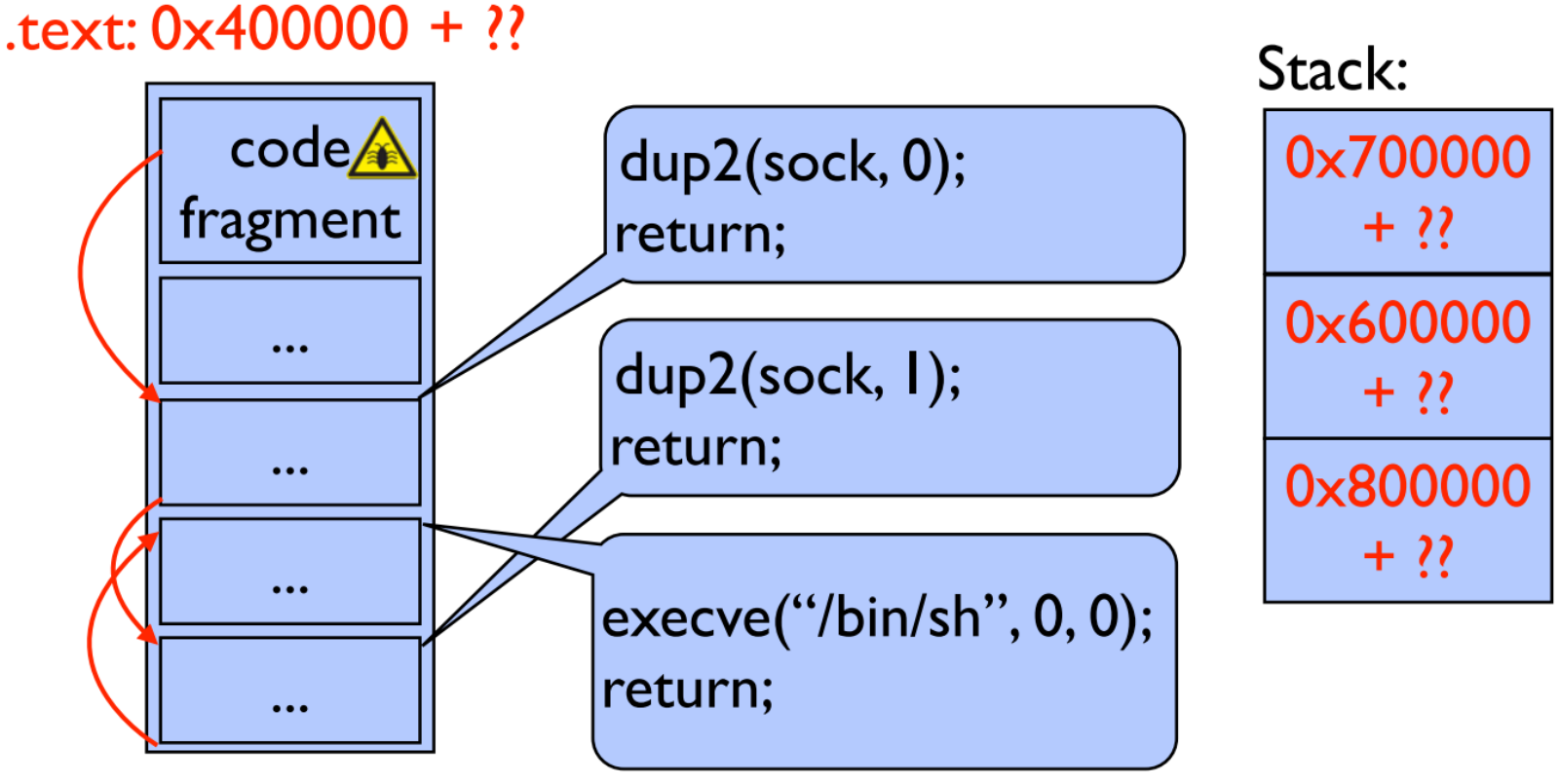
Return-Oriented Programming (ROP)



Address Space Layout Randomization (ASLR)



Address Space Layout Randomization (ASLR)



Exploit requirements today

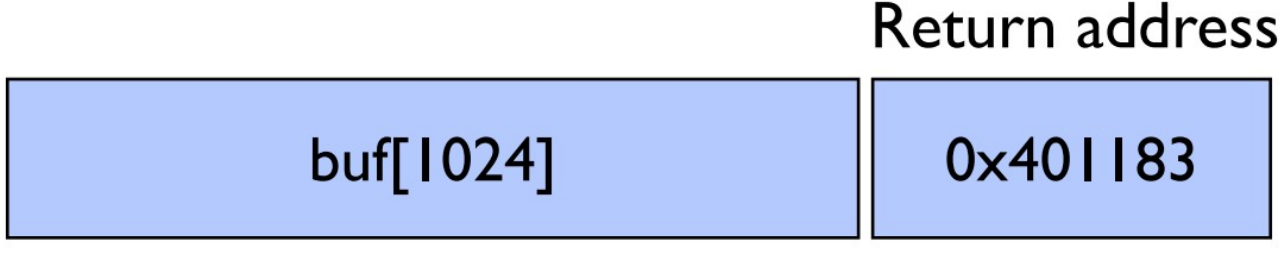
1. Break ASLR.
2. Copy of binary (find ROP gadgets / break NX).
    - Is it even possible to hack unknown applications?

Blind Return-Oriented Programming (BROP)

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.

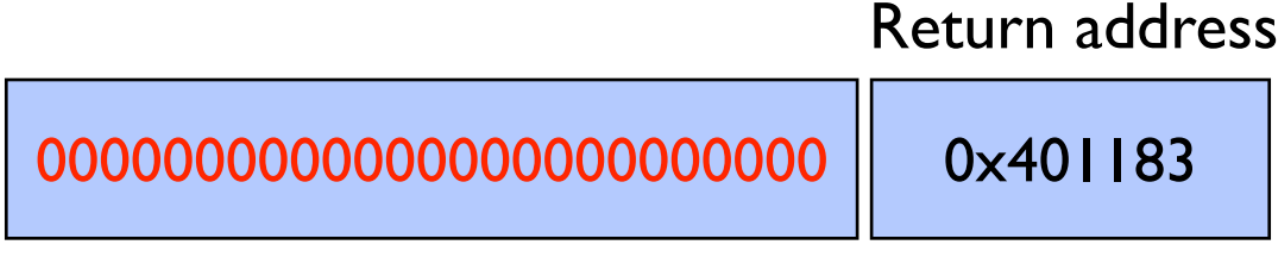
Defeating ASLR: stack reading

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



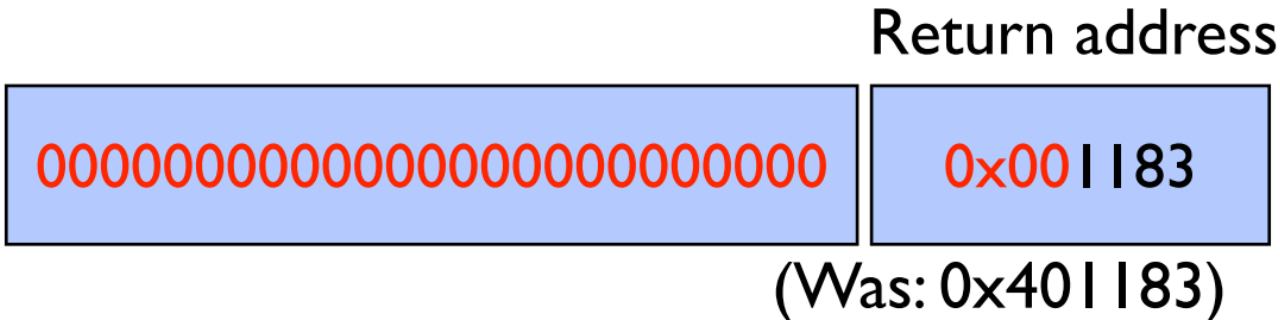
Defeating ASLR: stack reading

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



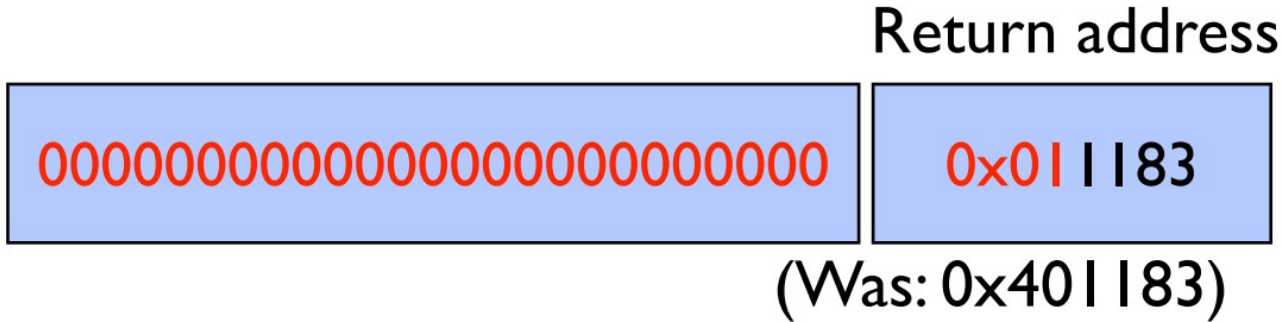
Defeating ASLR: stack reading

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



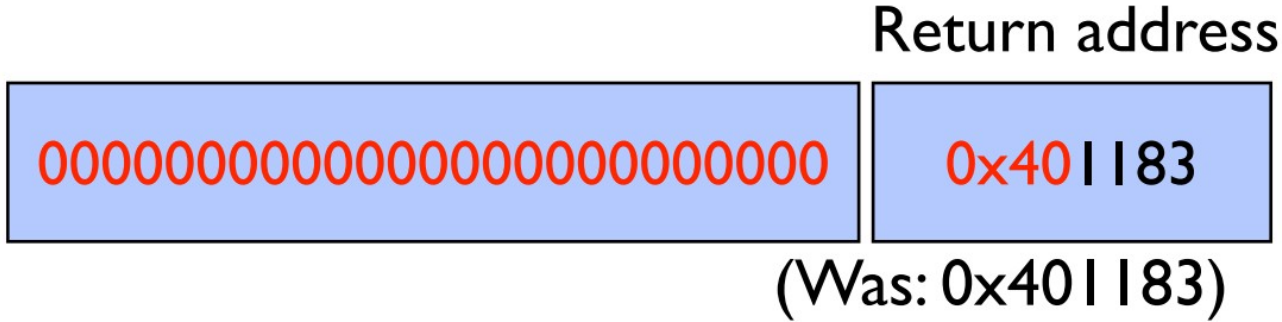
Defeating ASLR: stack reading

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

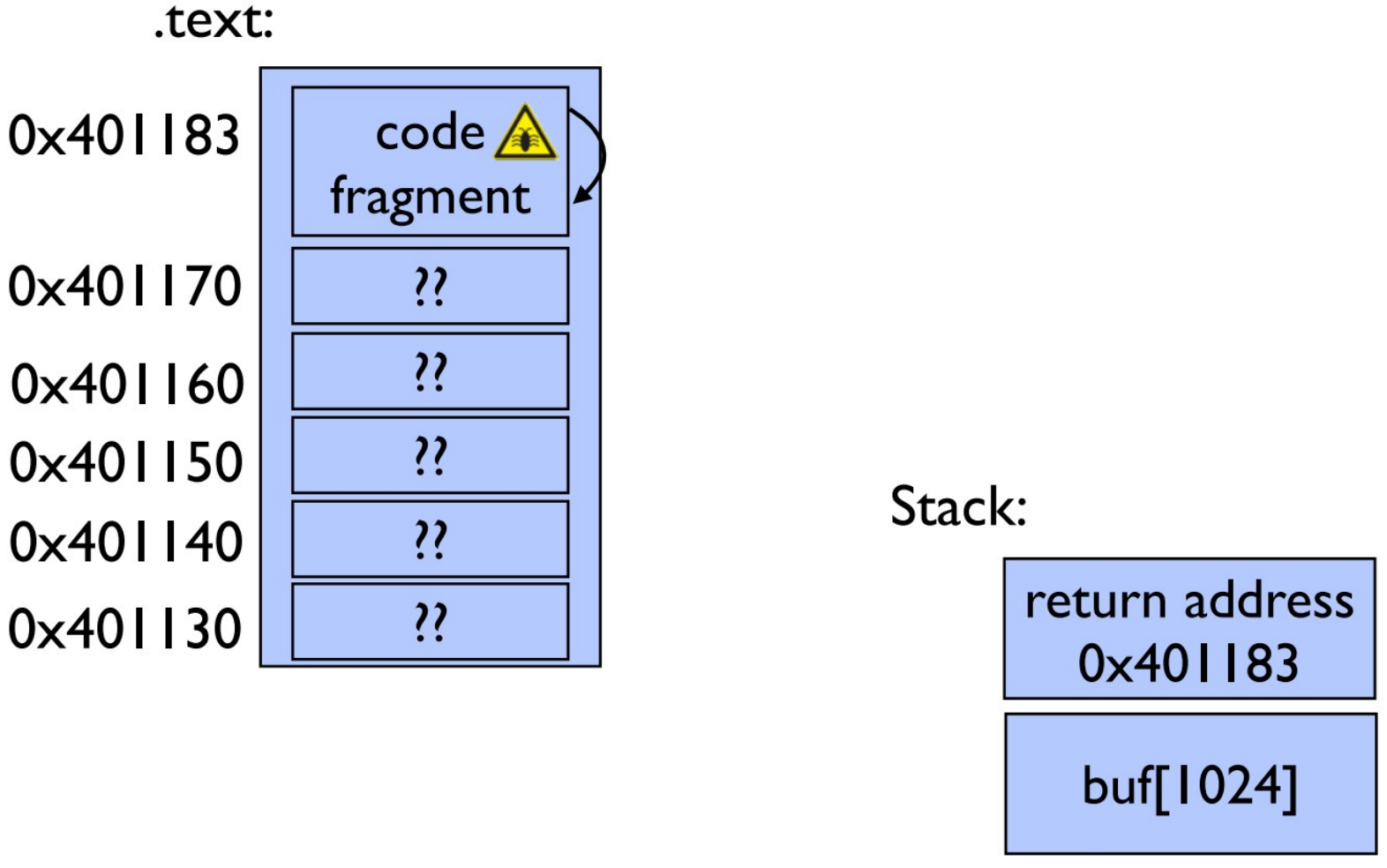


Defeating ASLR: stack reading

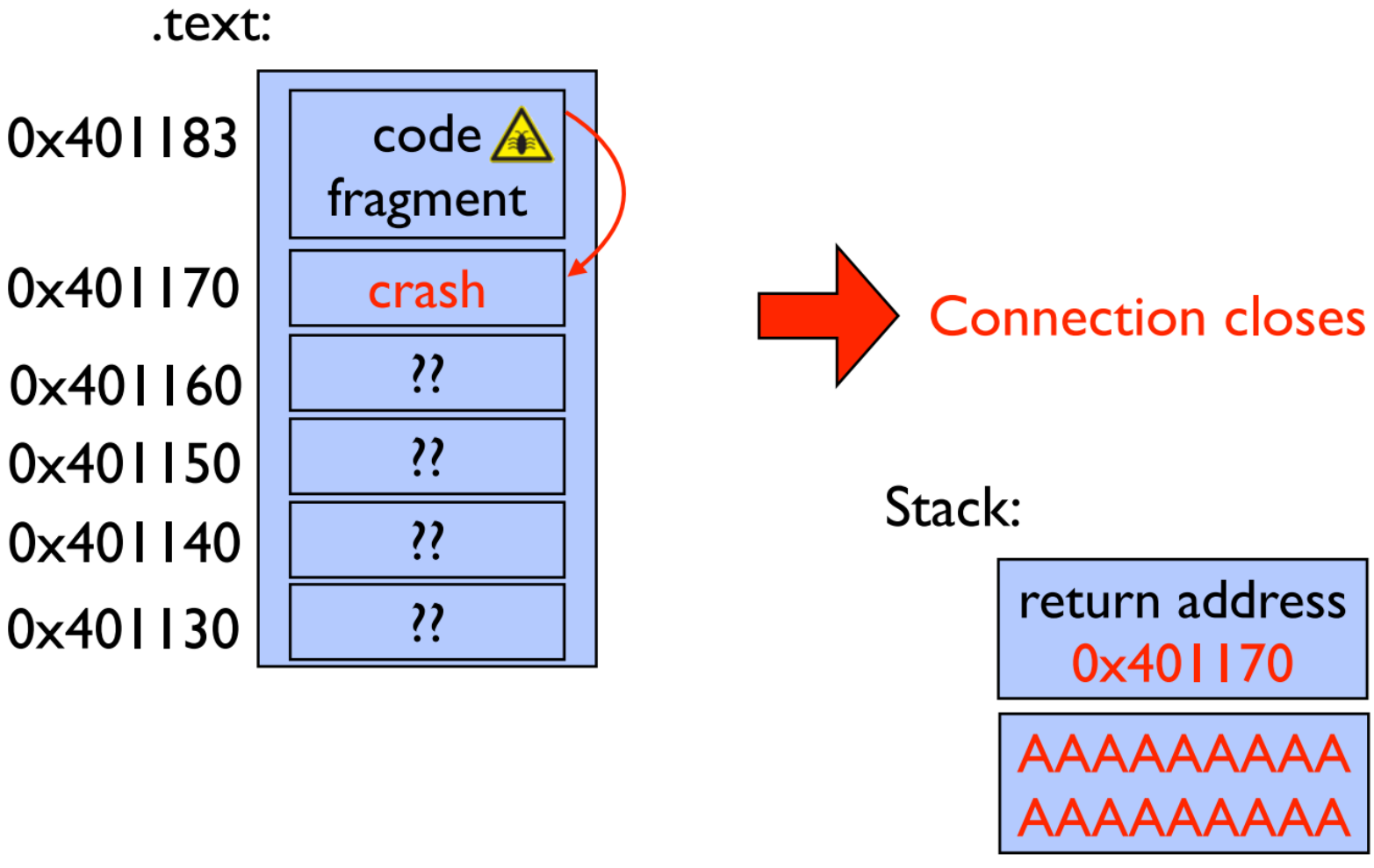
- Overwrite a single byte with value X:
  - No crash: stack had value X.
  - Crash: guess X was incorrect.
- Known technique for leaking canaries.



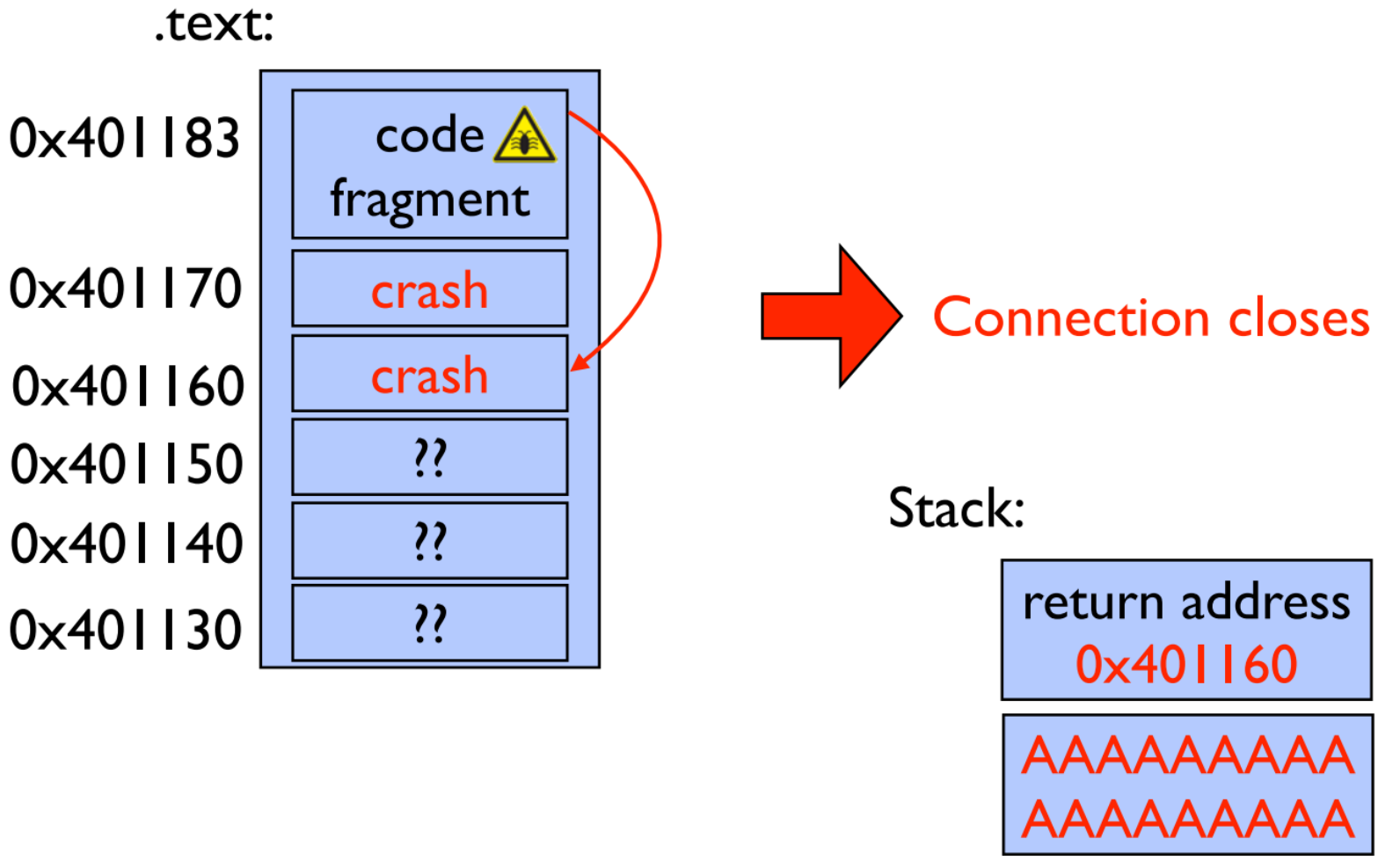
How to find gadgets?



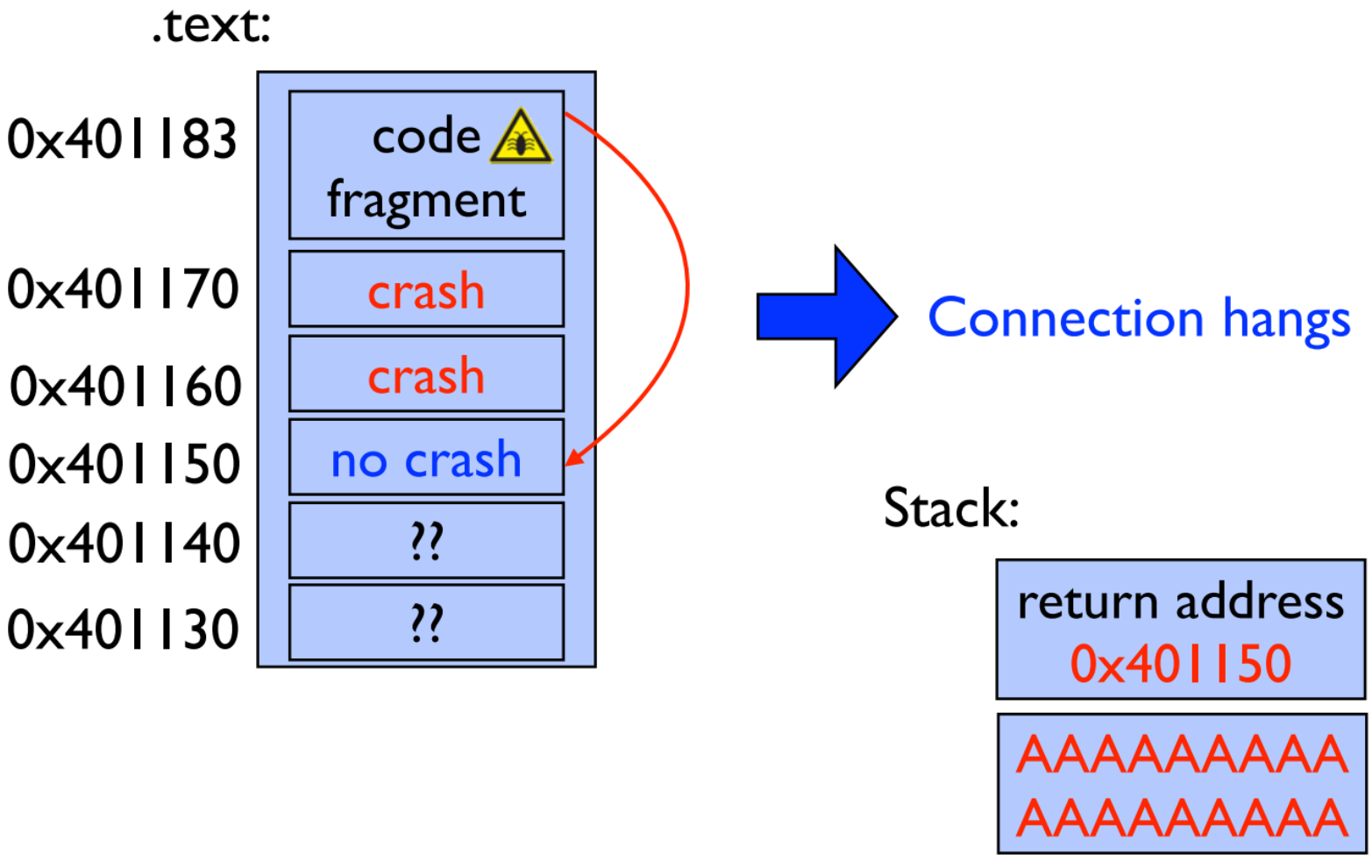
How to find gadgets?



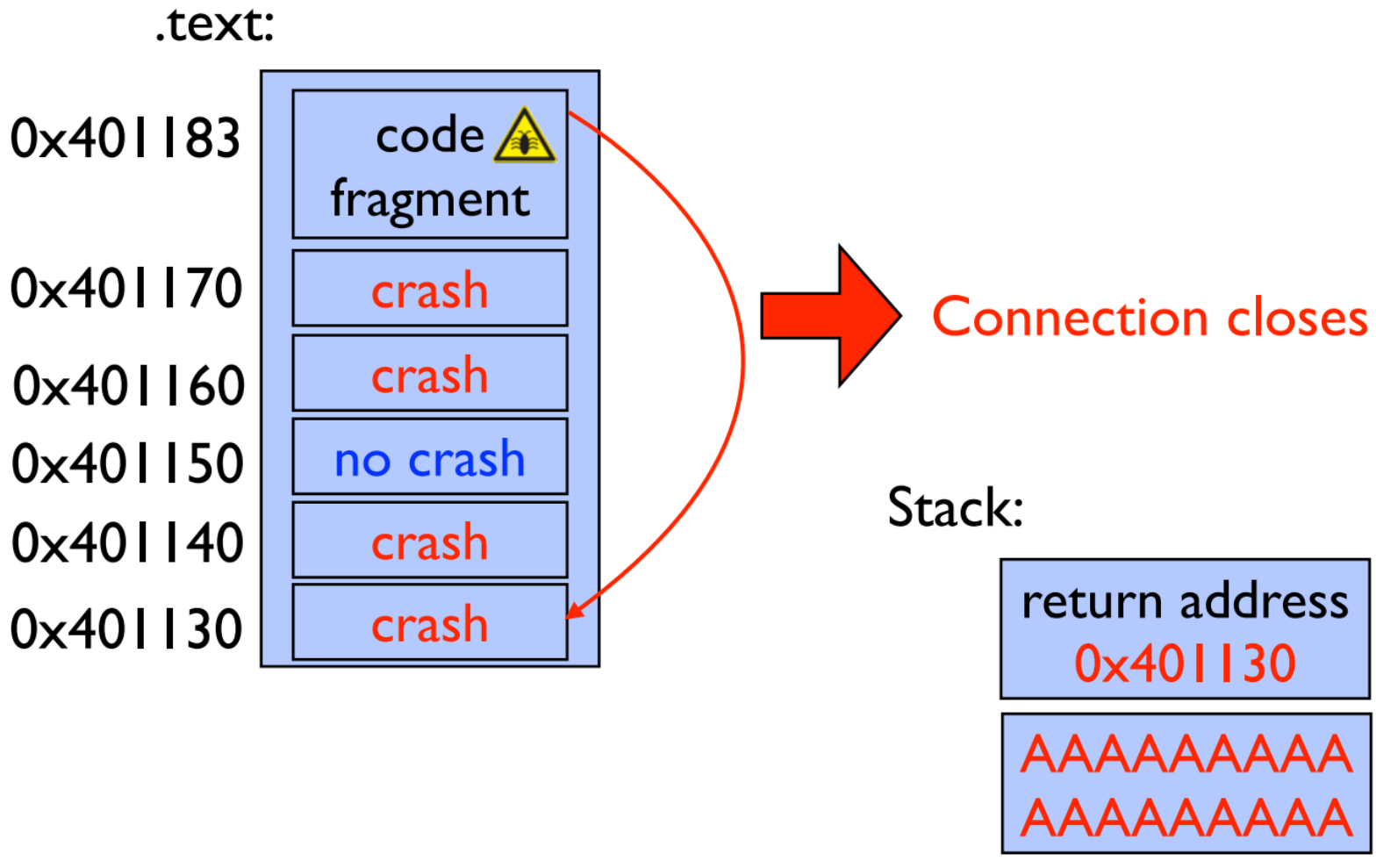
How to find gadgets?



How to find gadgets?



How to find gadgets?



Three types of gadgets

Stop gadget

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

- Never crashes

Crash gadget

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

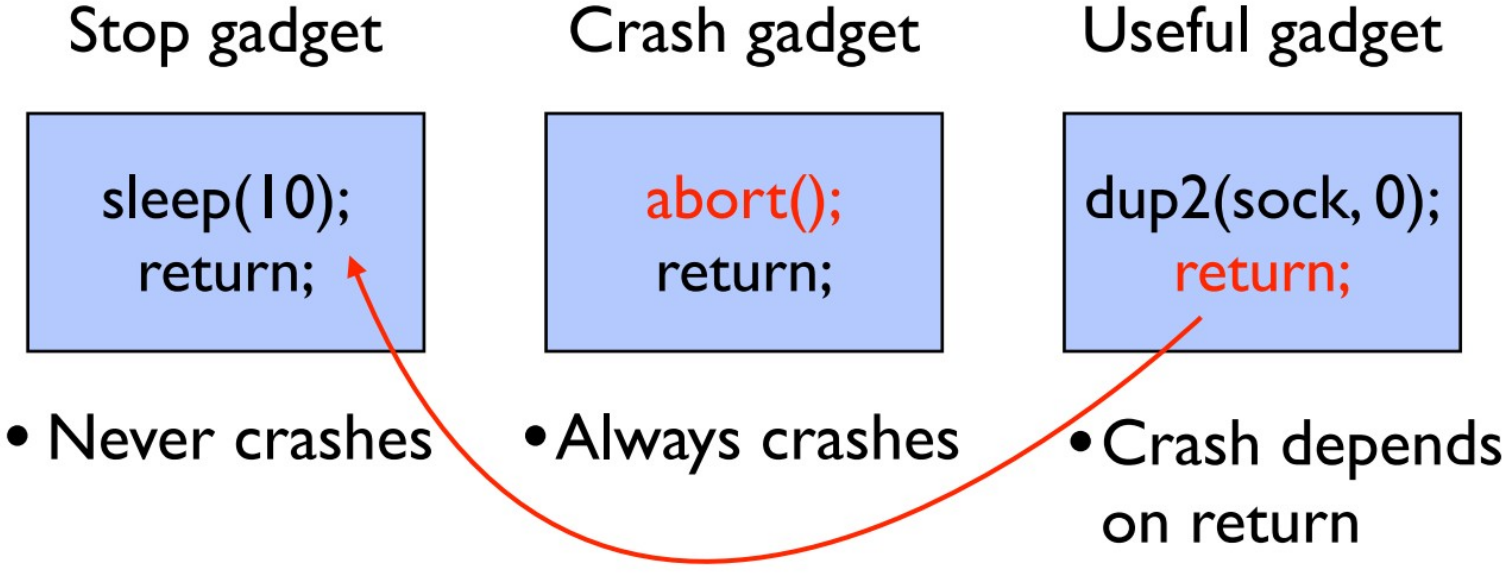
- Always crashes

Useful gadget

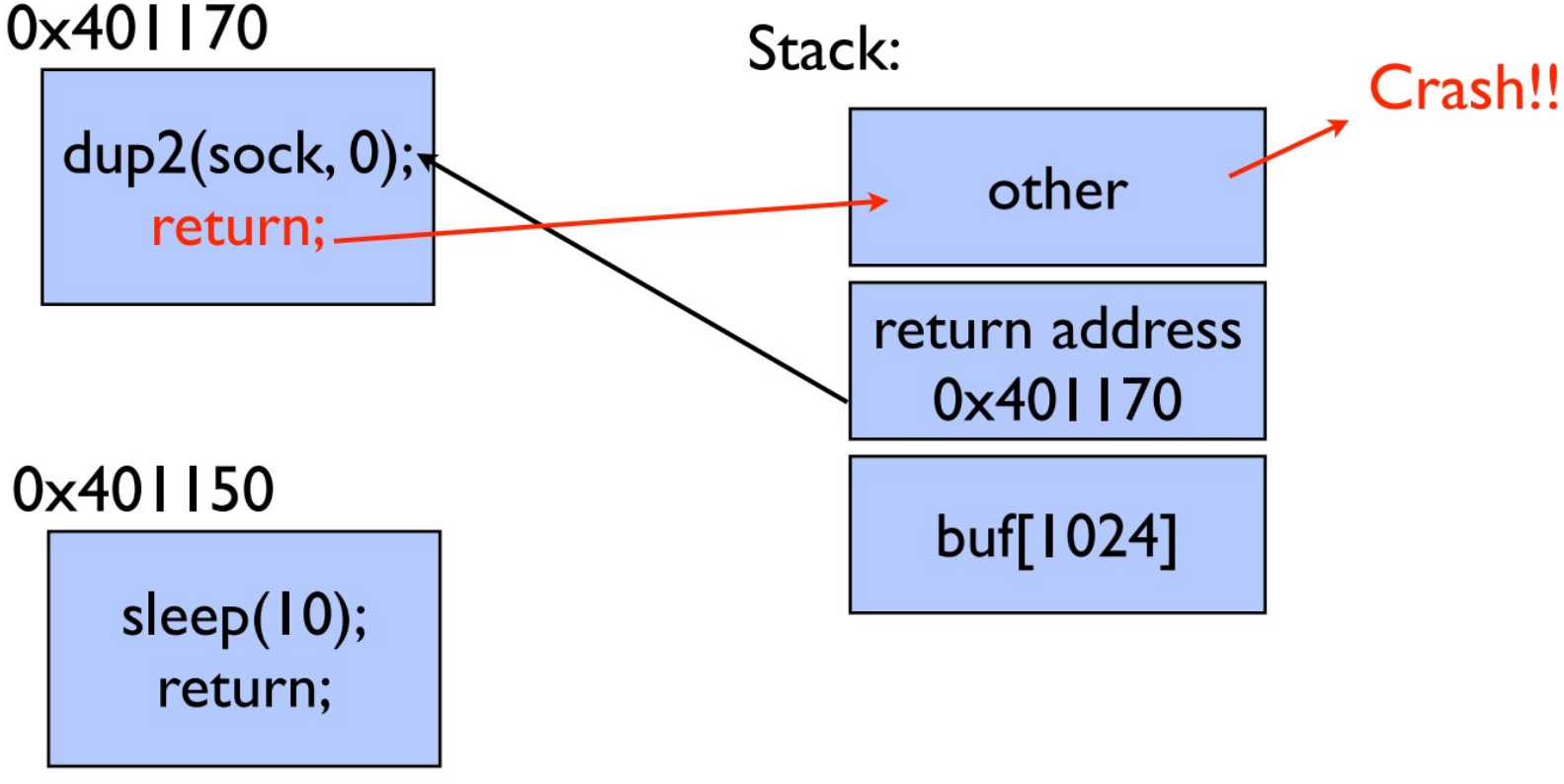
```
dup2(sock, 0);
return;
```

- Crash depends on return

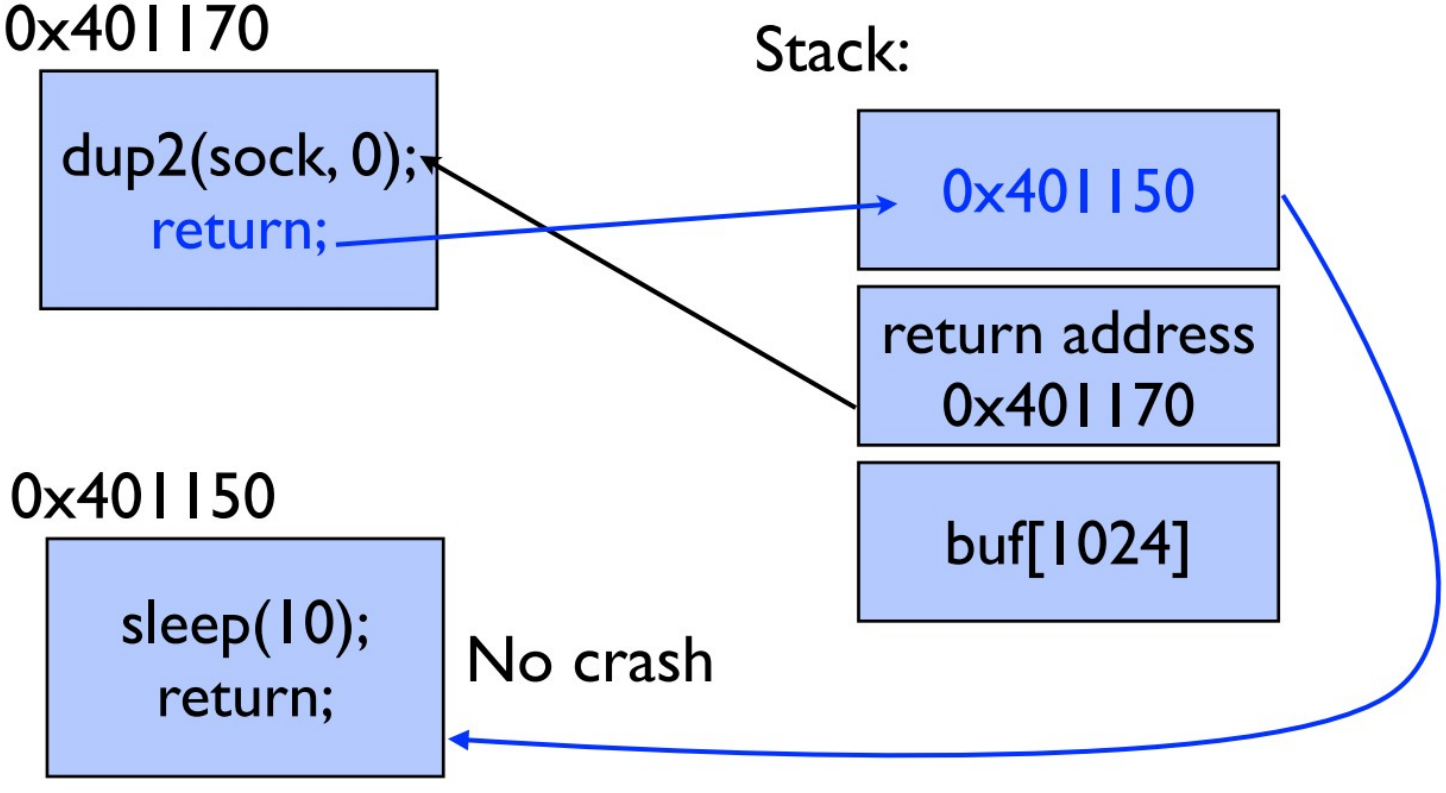
Three types of gadgets



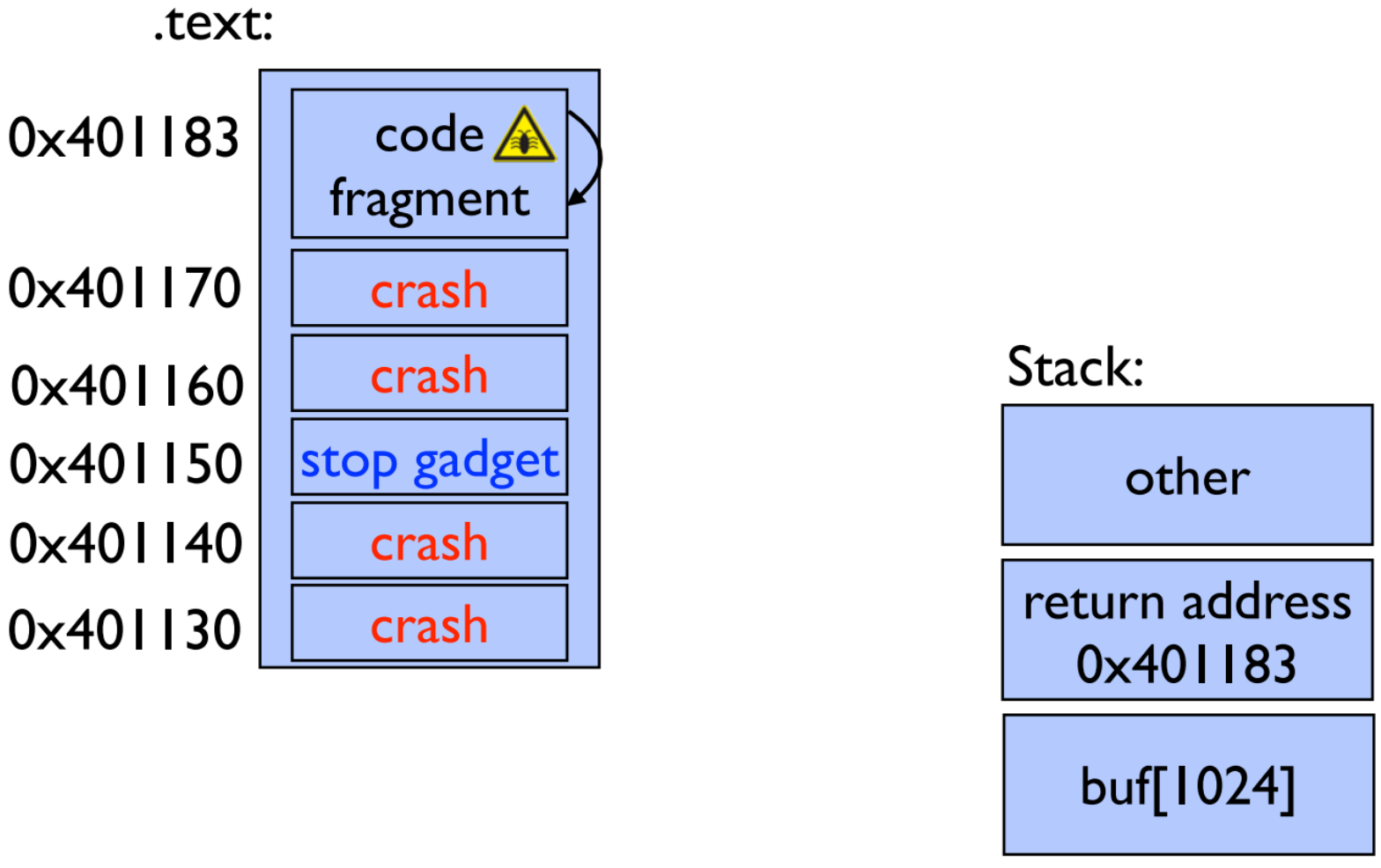
Finding useful gadgets



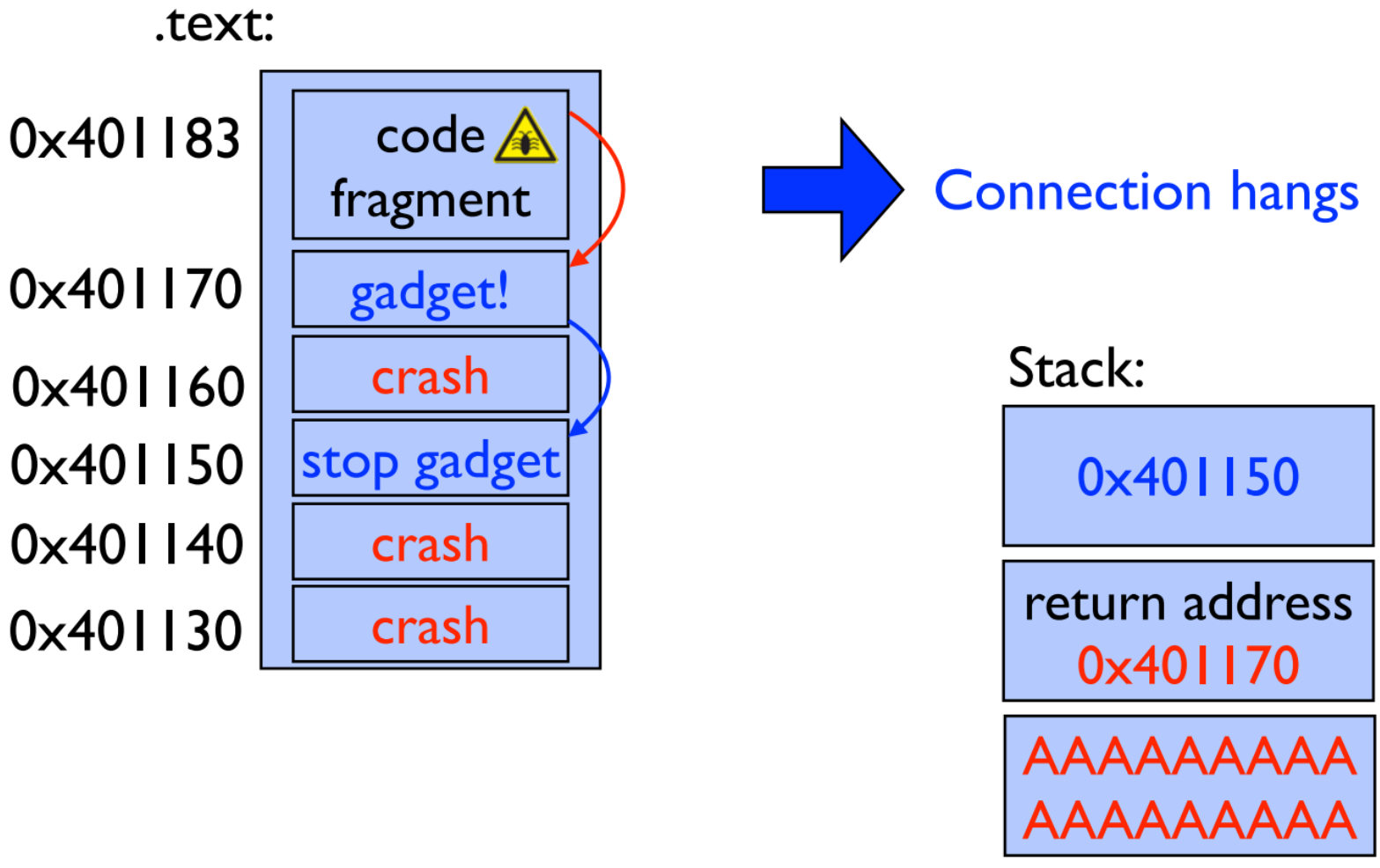
Finding useful gadgets



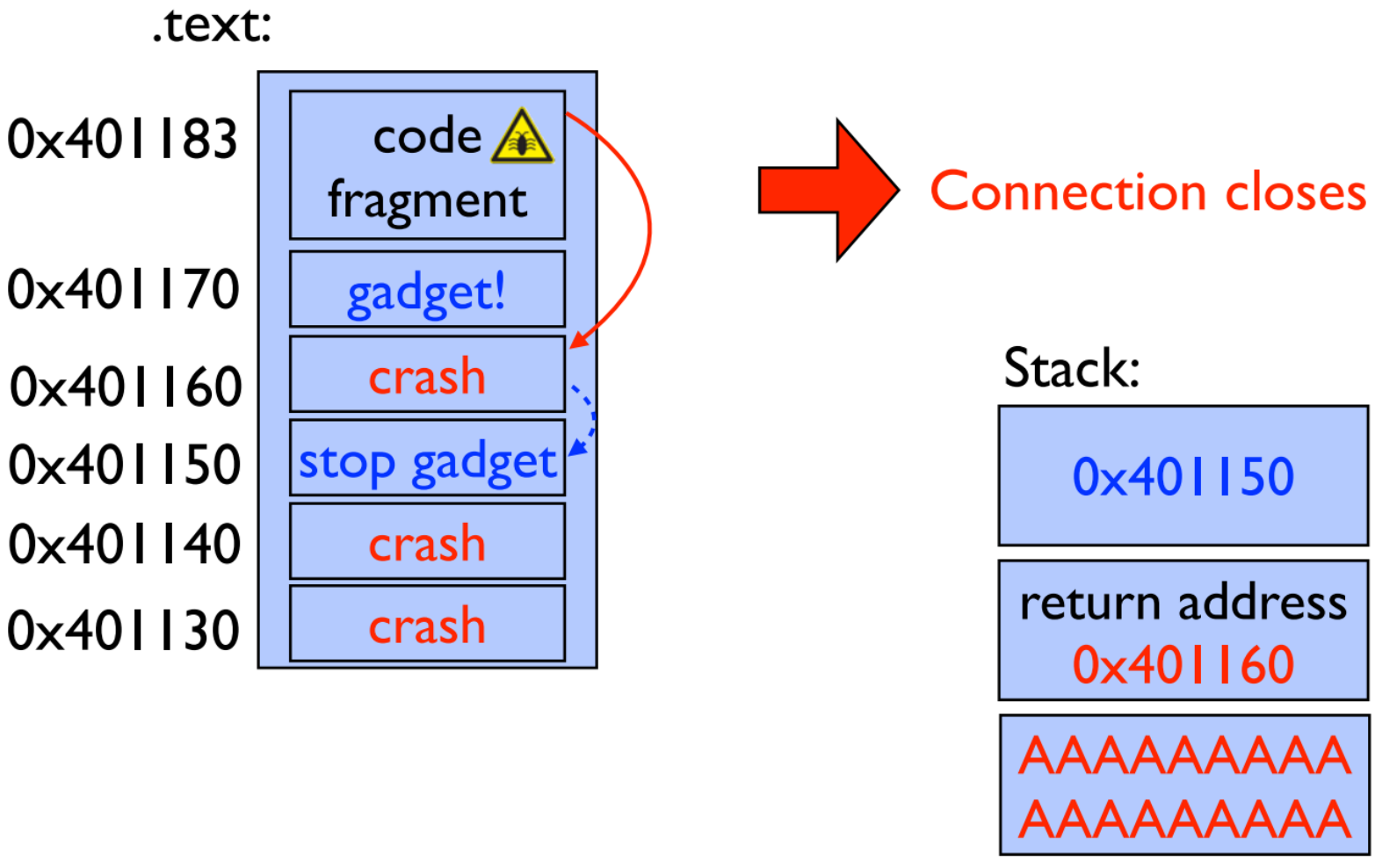
How to find gadgets?



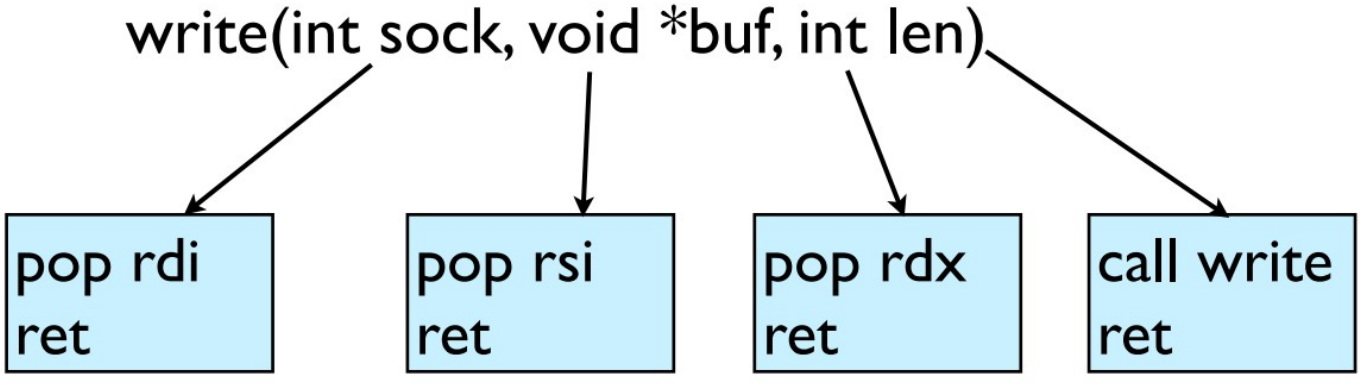
How to find gadgets?



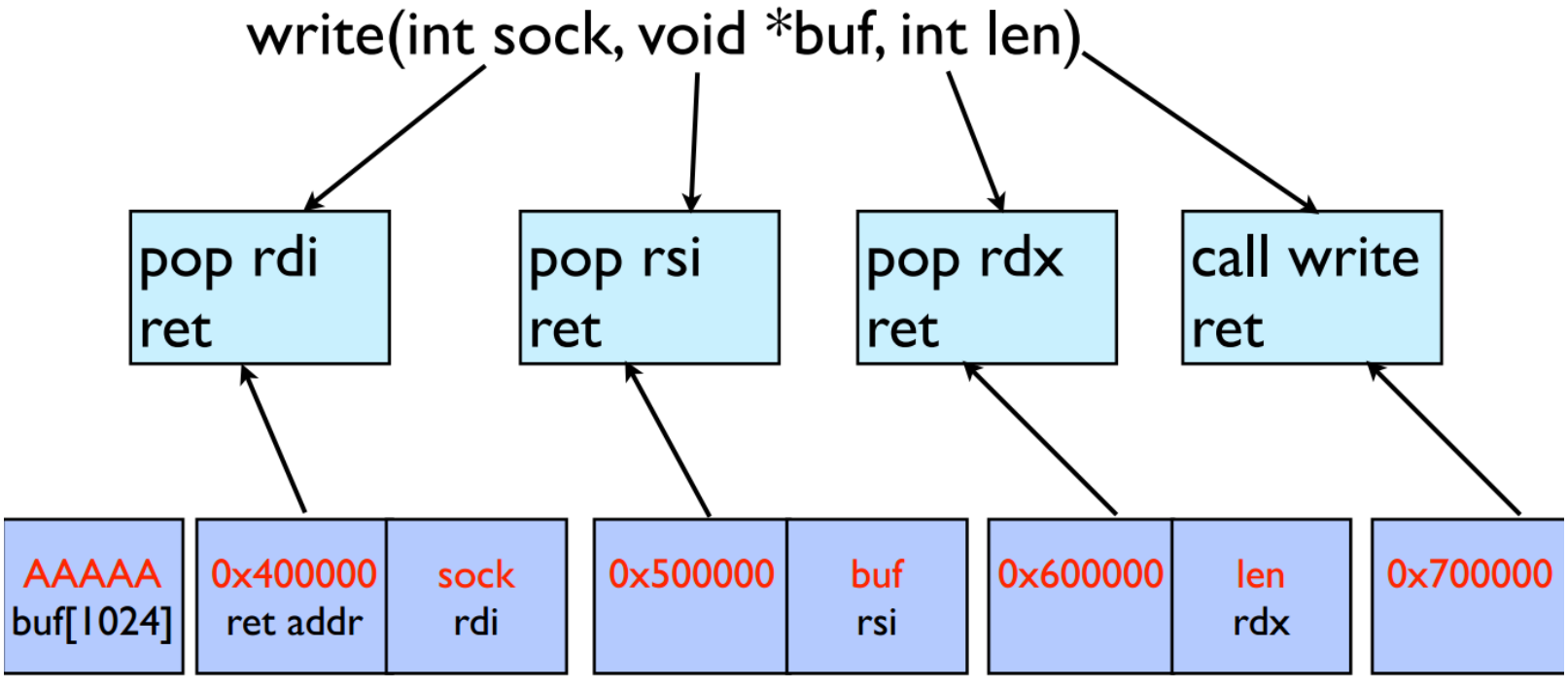
How to find gadgets?



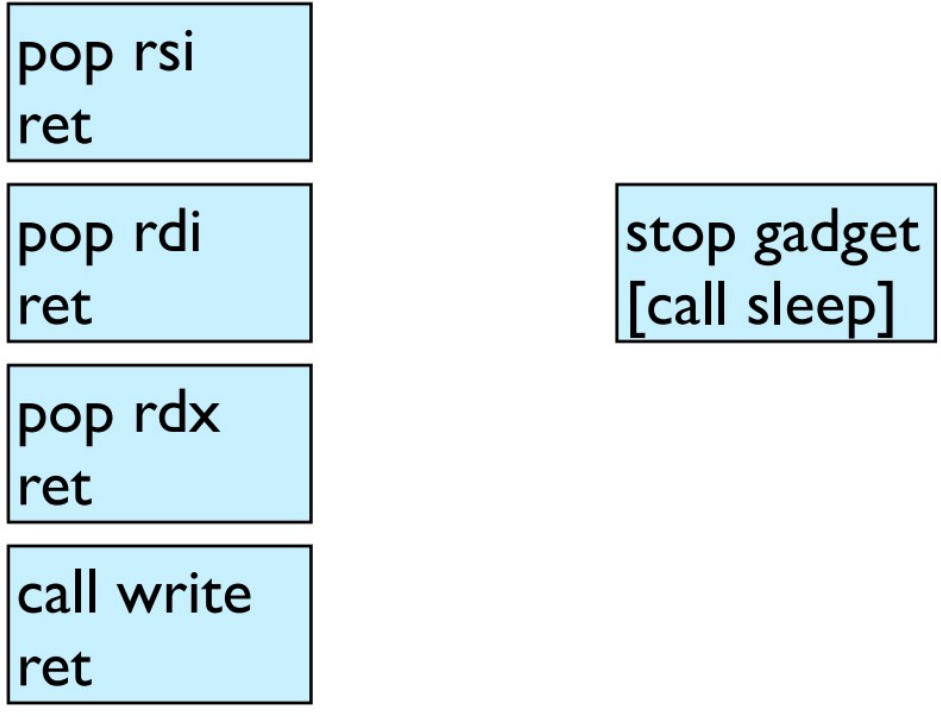
What are we looking for?



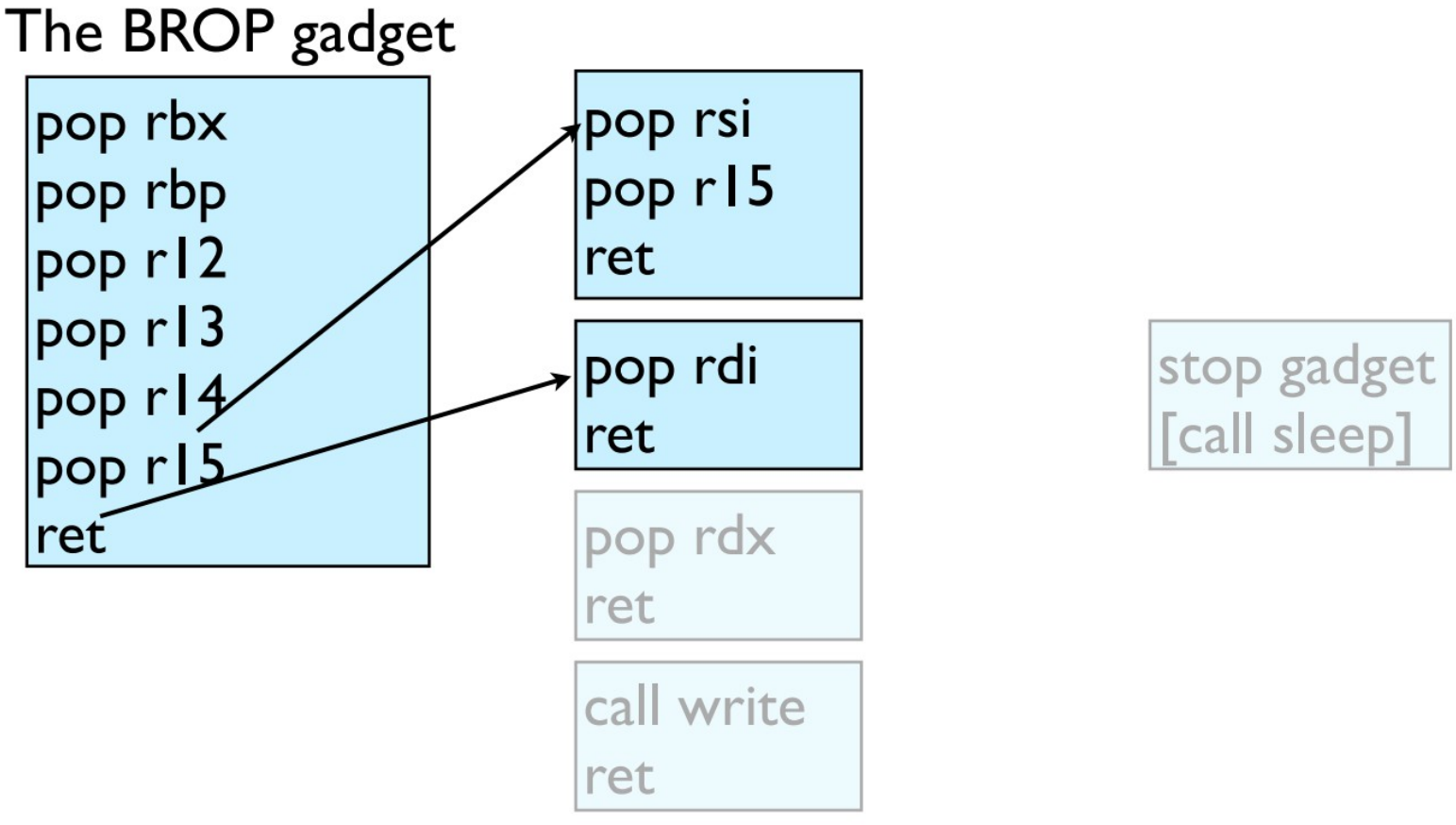
What are we looking for?



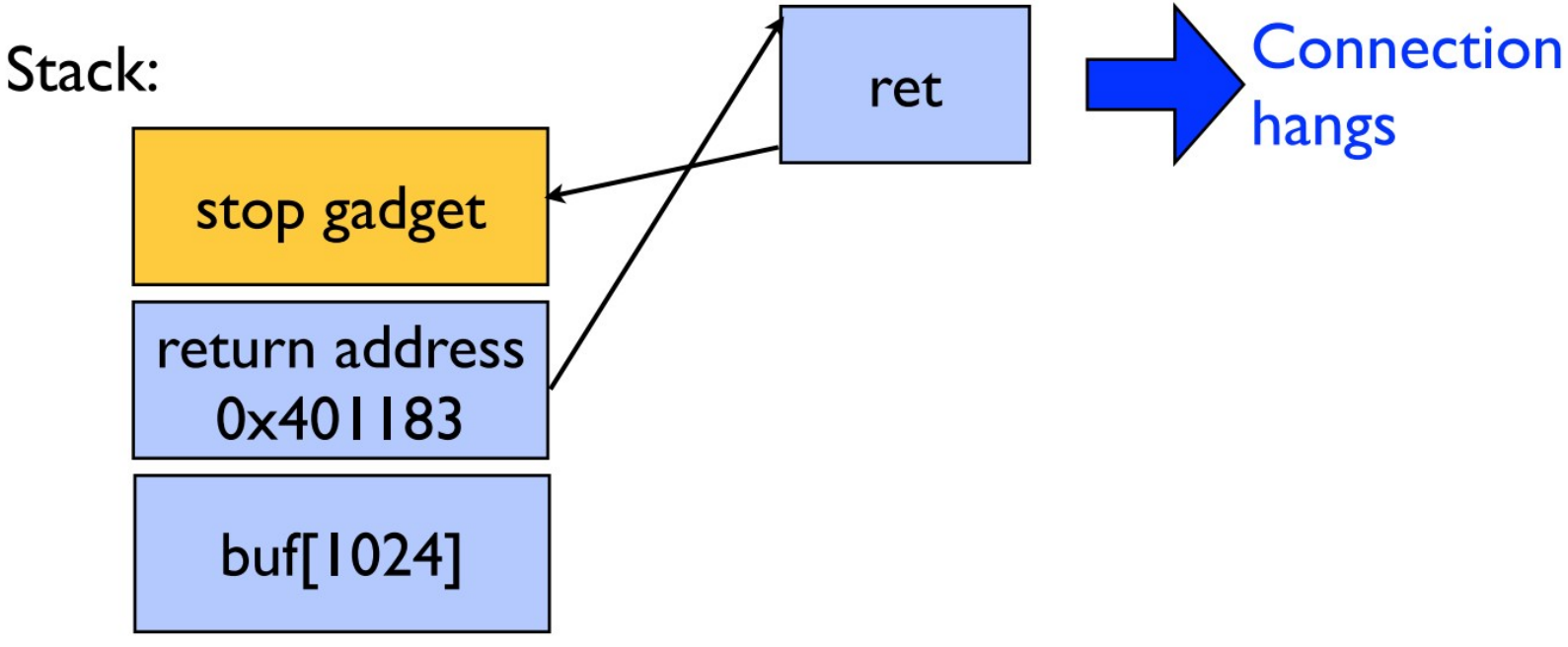
Pieces of the puzzle



Pieces of the puzzle

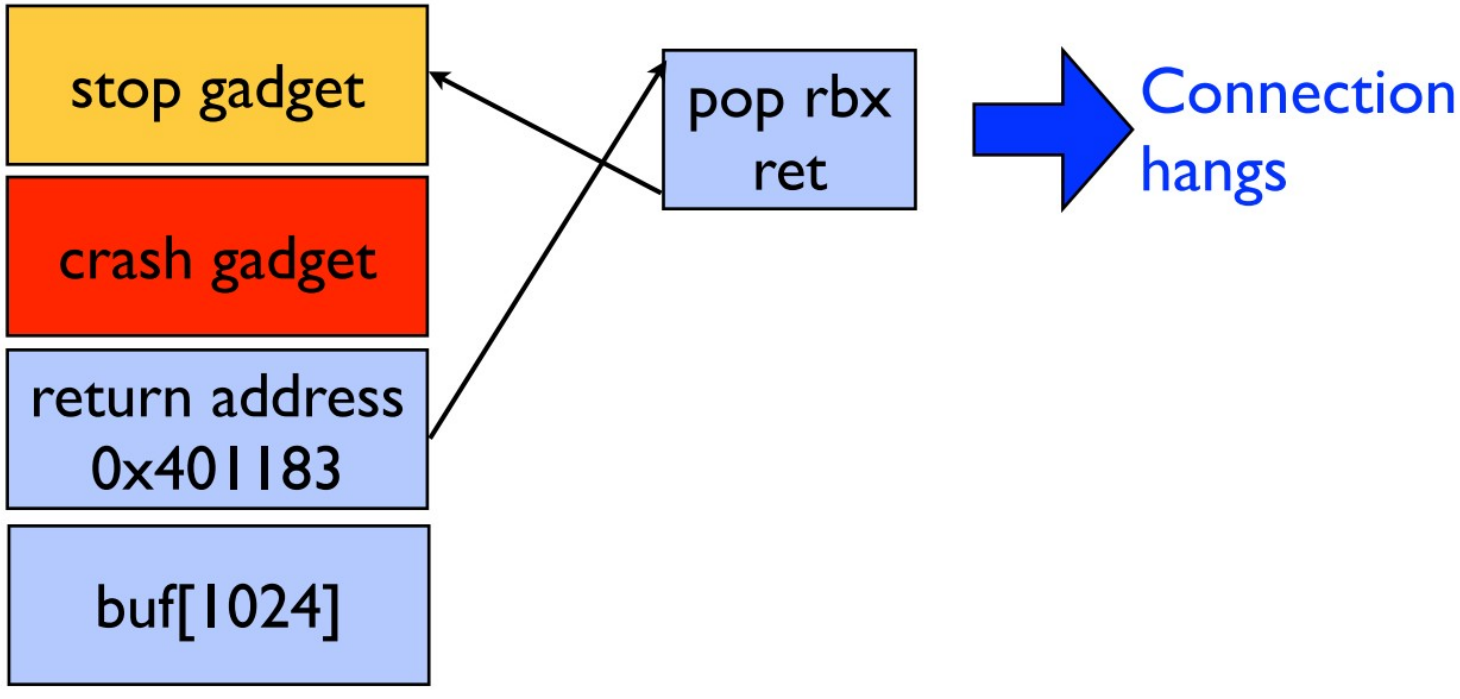


Finding the BROP gadget

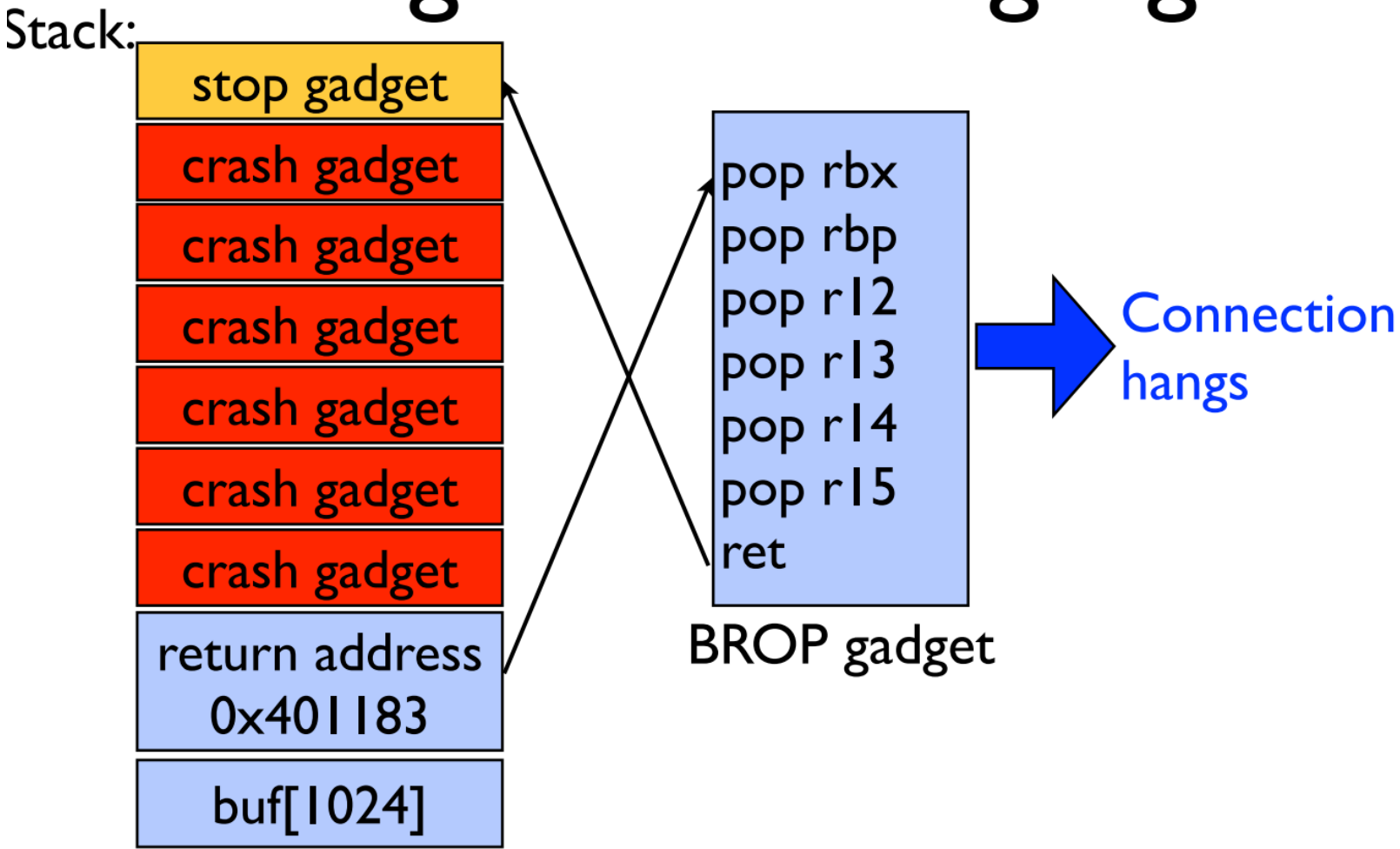


Finding the BROP gadget

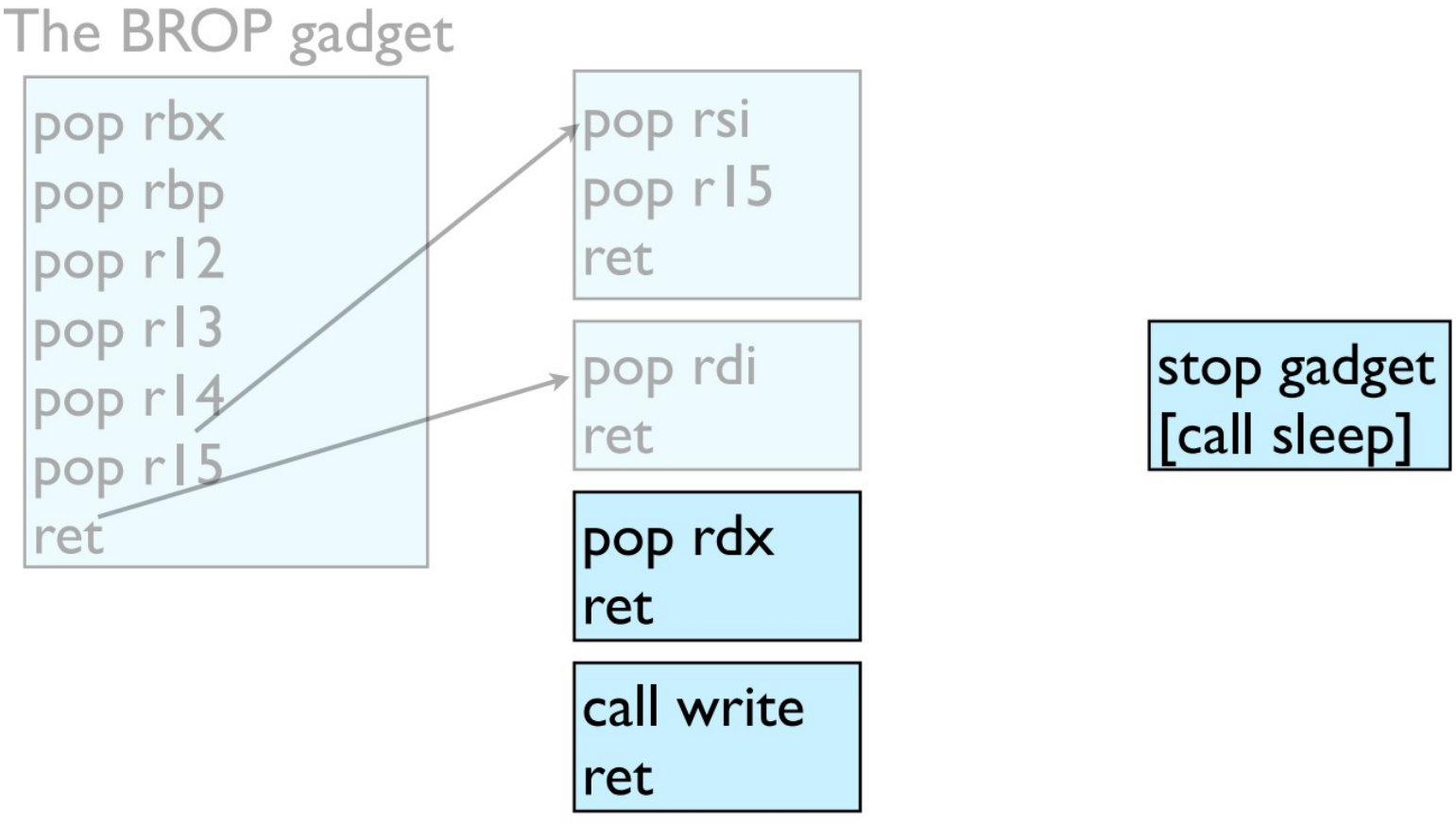
Stack:



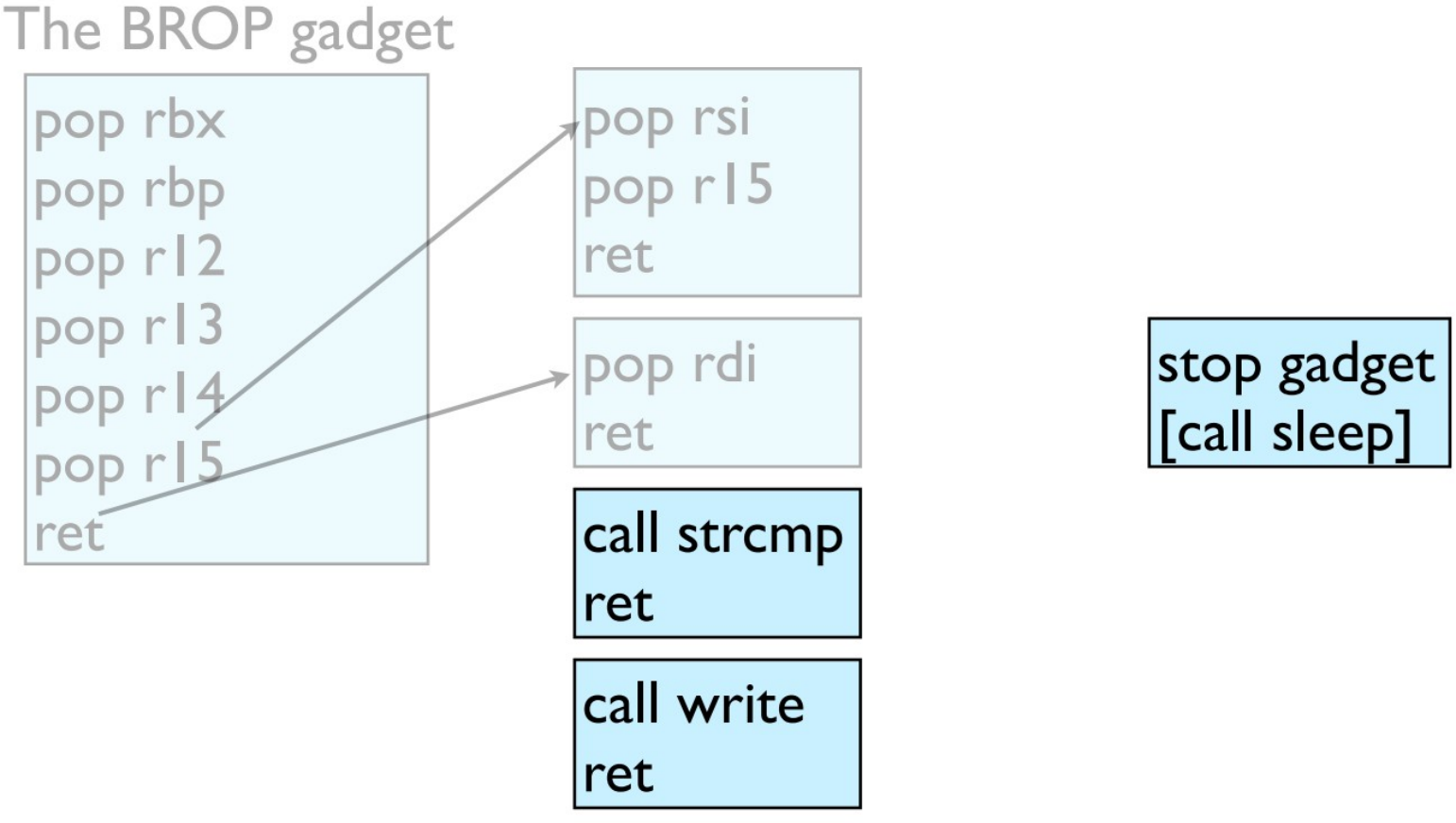
Finding the BROP gadget



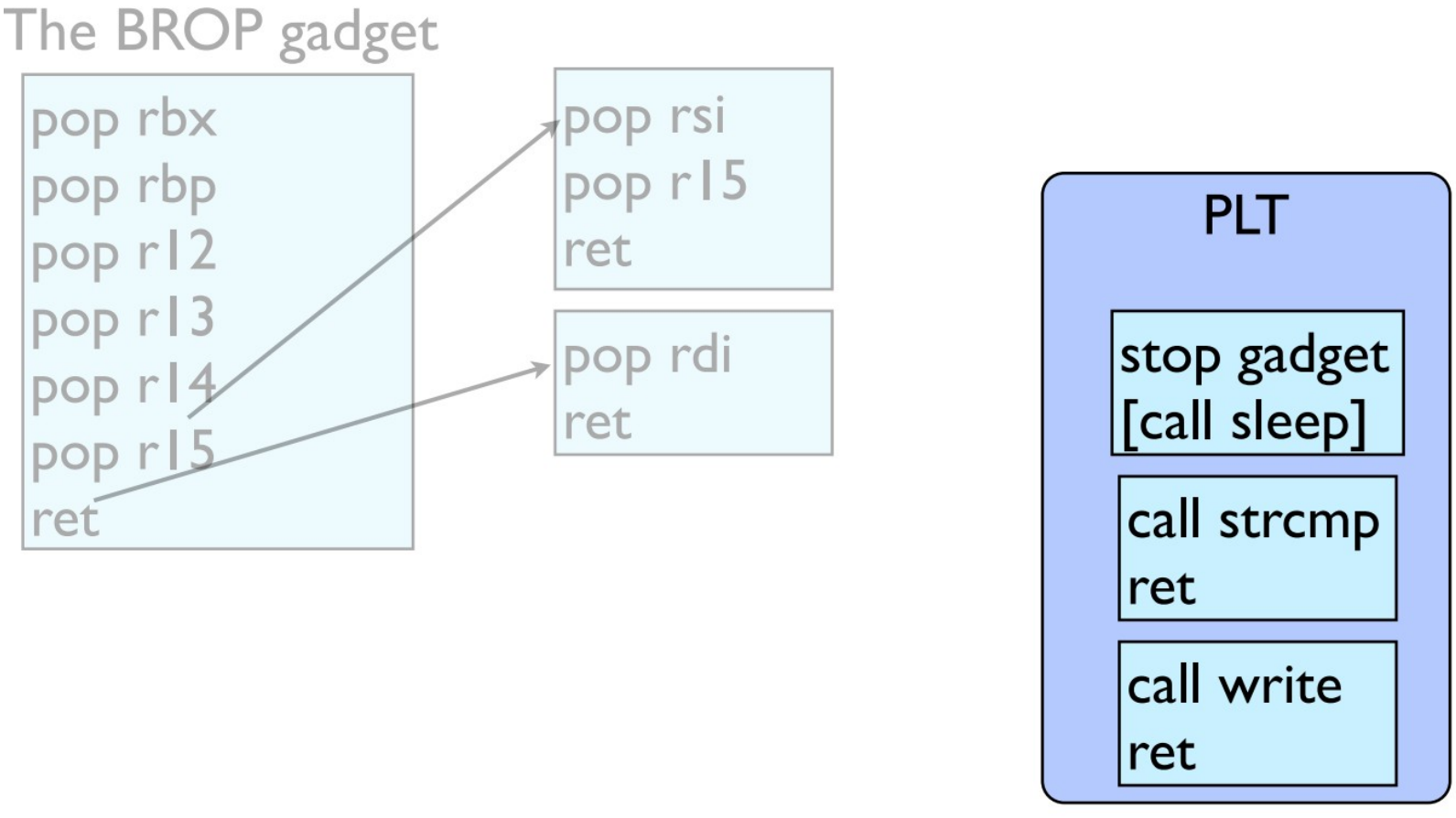
Pieces of the puzzle



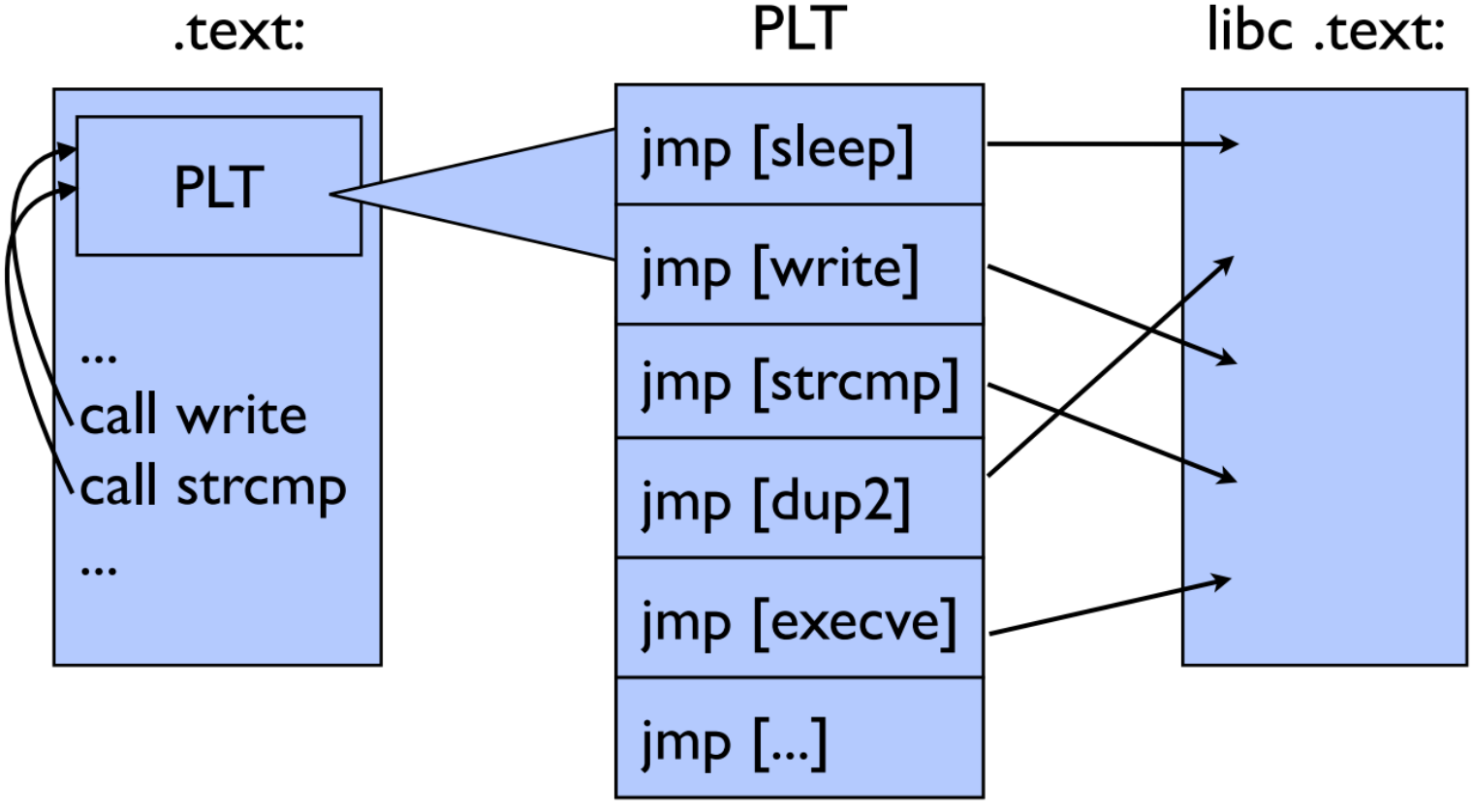
Pieces of the puzzle



Pieces of the puzzle



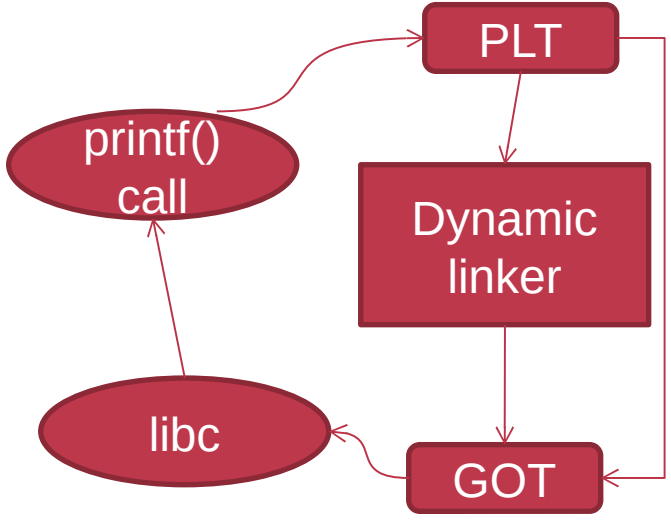
Procedure Linking Table (PLT)

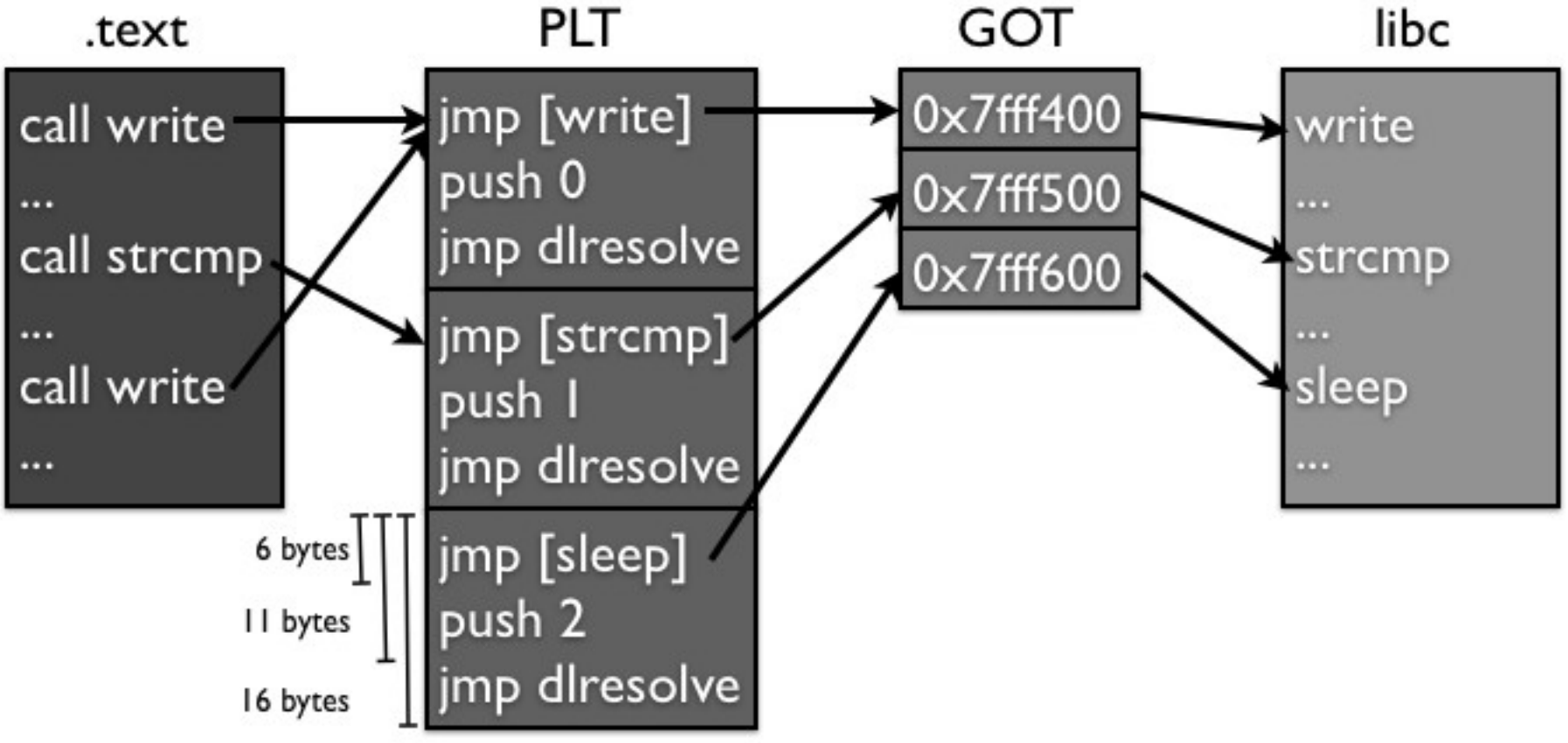


GOT & PLT

Jump to the address that this entry of the GOT contains:

- First time: call to dynamic linker and write the function address to GOT
- Not first time: the GOT already contains the address that points to printf.




.text
PLT
GOT
libc
call write
...
call strcmp
...
call write
...
jmp [write]
push 0
jmp dlresolve
jmp [strcmp]
push 1
jmp dlresolve
jmp [sleep]
push 2
jmp dlresolve
0x7fff400
0x7fff500
0x7fff600
write
...
strcmp
...
sleep
...
6 bytes
11 bytes
16 bytes

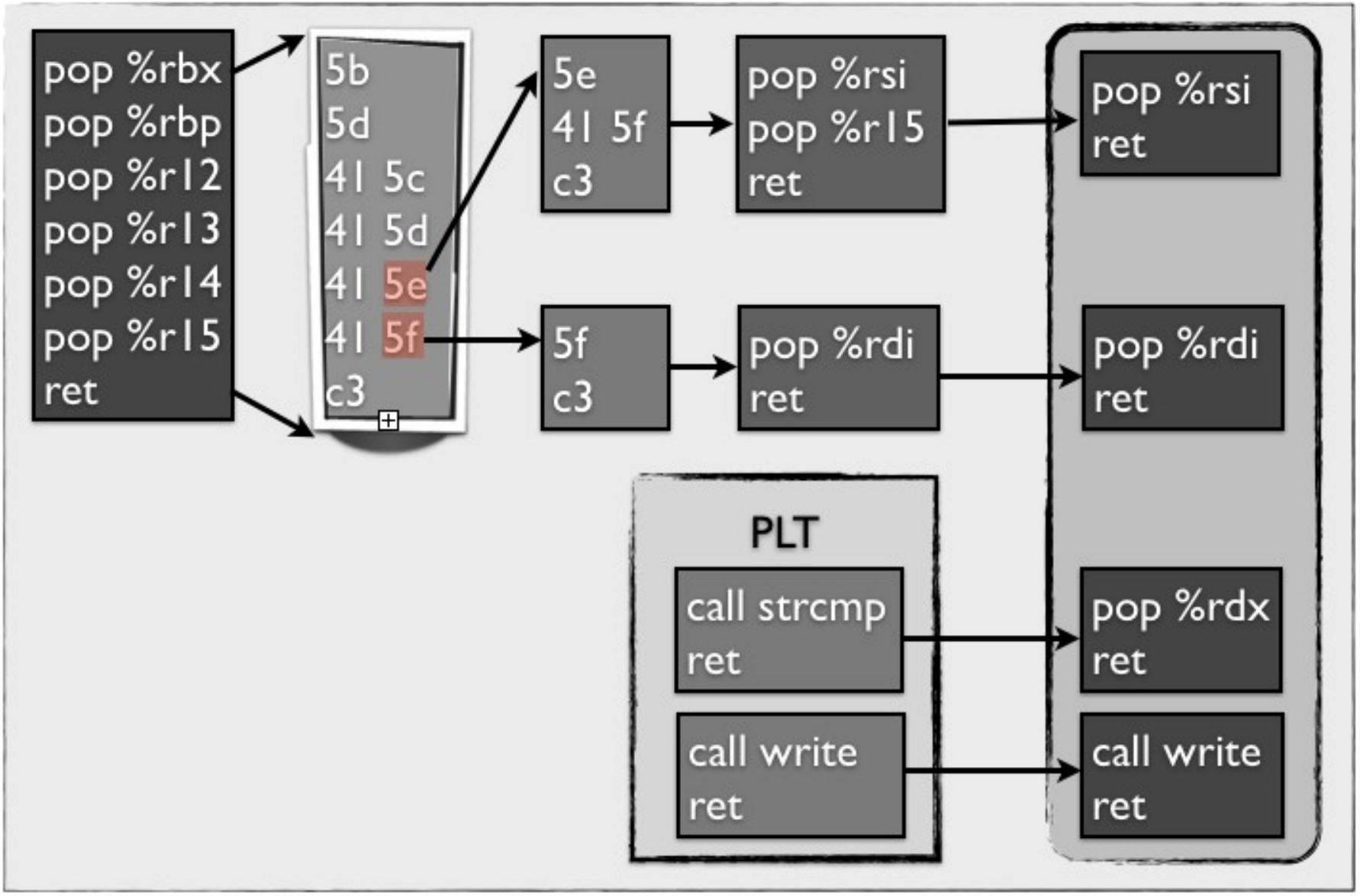
Fingerprinting strcmp

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

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

Finding write

- Try sending data to socket by calling candidate PLT function.
- check if data received on socket.
- chain writes with different FD numbers to find socket. Use multiple connections.


pop %rbx
pop %rbp
pop %r12
pop %r13
pop %r14
pop %r15
ret
5b
5d
41 5c
41 5d
41 5e
41 5f
c3
5e
41 5f
c3
pop %rsi
pop %r15
ret
5f
c3
pop %rdi
ret
pop %rsi
ret
pop %rdi
ret
PLT
call strcmp
ret
call write
ret
pop %rdx
ret
call write
ret

Launching a shell

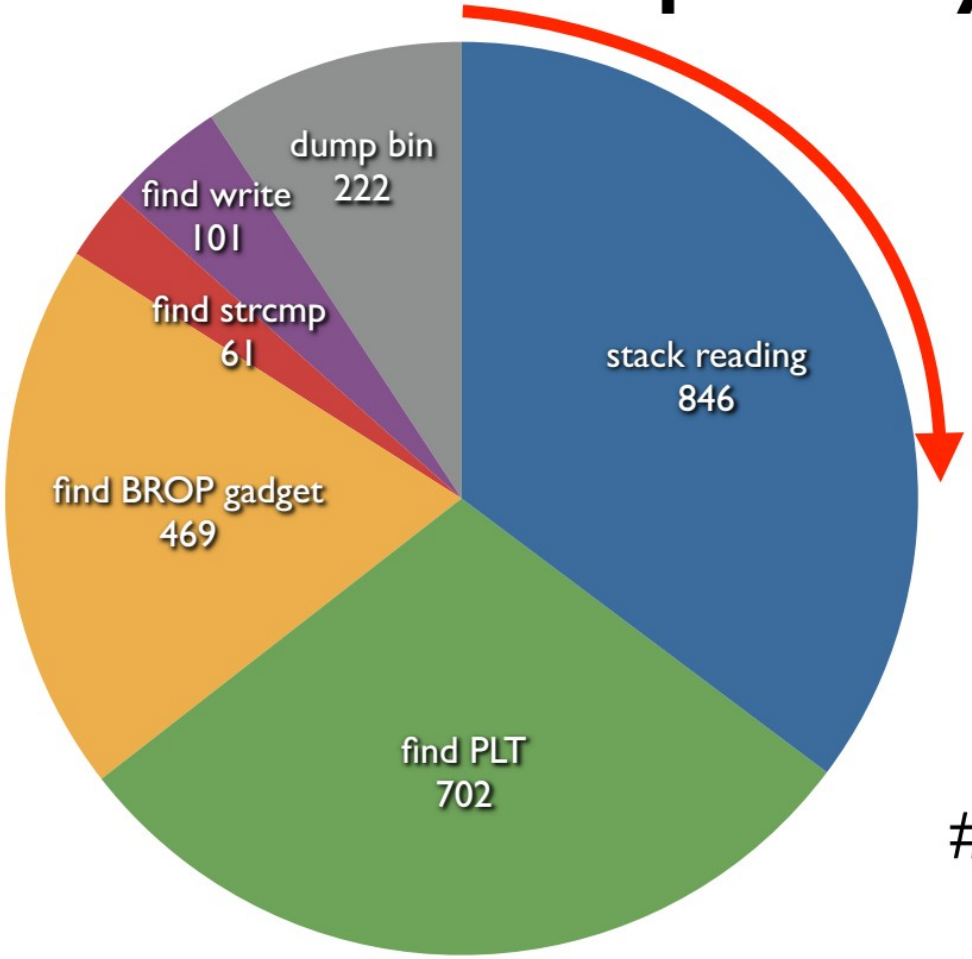
1. dump binary from memory to network. Not blind anymore!
2. dump symbol table to find PLT calls.
3. redirect stdin/out to socket:
   - dup2(sock, 0); dup2(sock, 1);
4. read() “/bin/sh” from socket to memory
5. execve(“/bin/sh”, 0, 0)

Braille

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

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

Attack complexity



# of requests
for nginx

Try It!

http://www.scs.stanford.edu/brop/

More information

- S&P'14 paper: http://www.scs.stanford.edu/brop/bittau-brop.pdf
- Blog: http://drops.wooyun.org/tips/3071