Advanced Exploitation

Or why all your defenses won't stop smart attackers

Herbert Bos

VU University Amsterdam

VUSec

Systems Security

VUSec

Three Focus Areas

Defenses

Binary & Malware Analysis

Attacks

2016

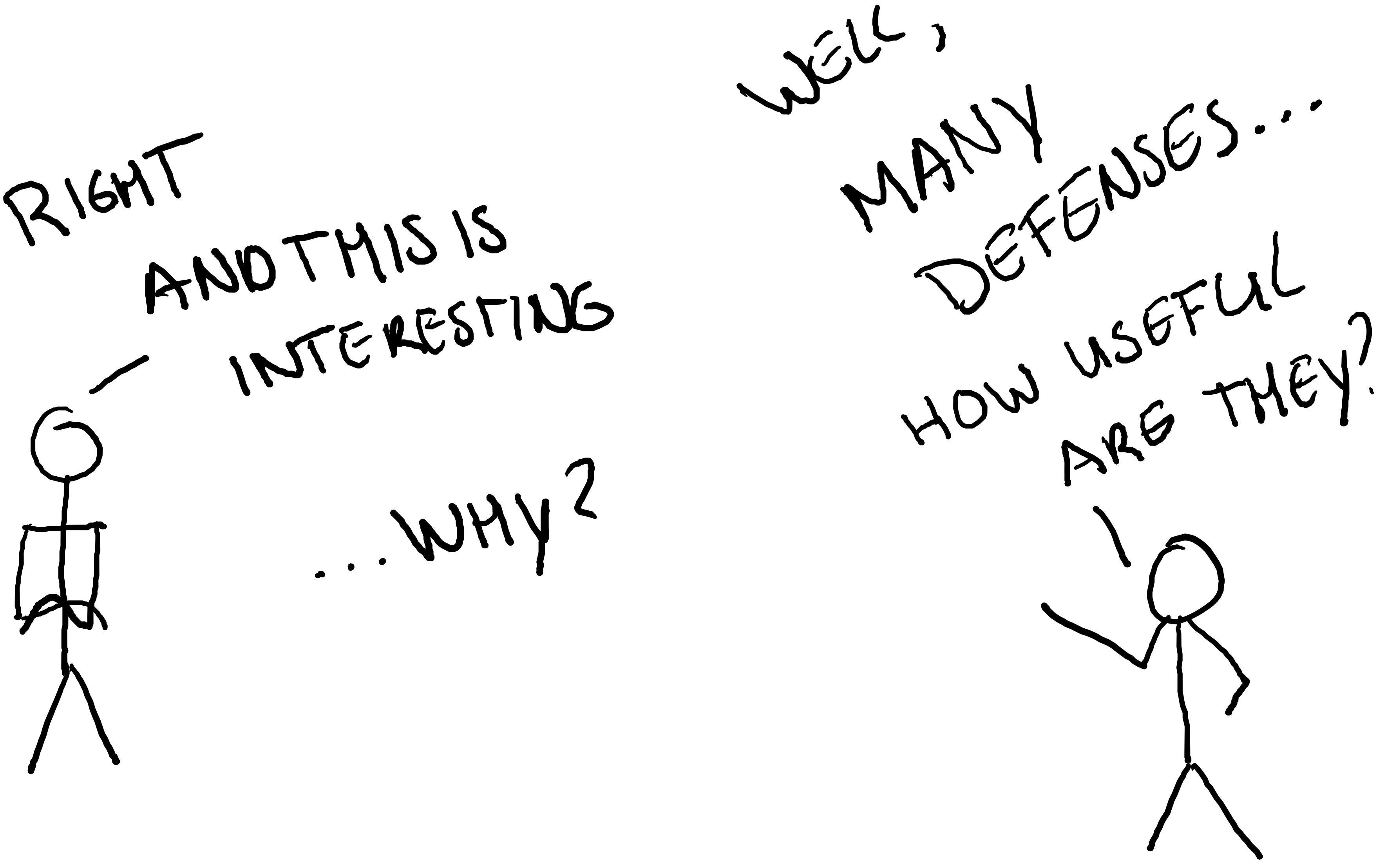
2017

2017

Security Bugs

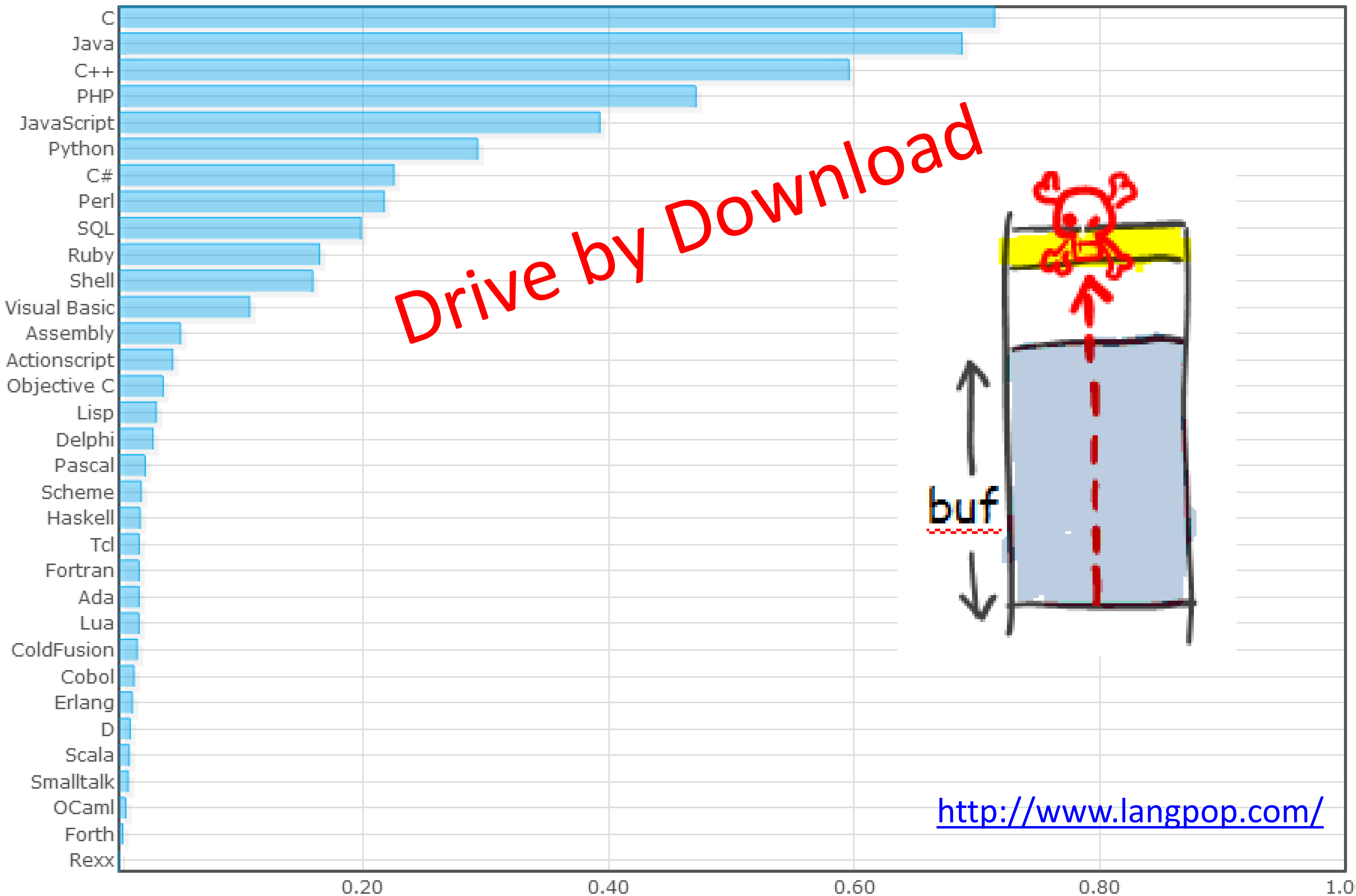
- Allow attackers to "take control"
  - Execute anything they want
- Turn the program into a "weird machine"
  - That can be programmed—in a weird way


RIGHT
AND THIS IS INTERESTING
...WHY?


RIGHT
AND THIS IS INTERESTING
...WHY?
WELL, MANY DEFENSES...
HOW USEFUL ARE THEY?

Focus
Memory Corruption

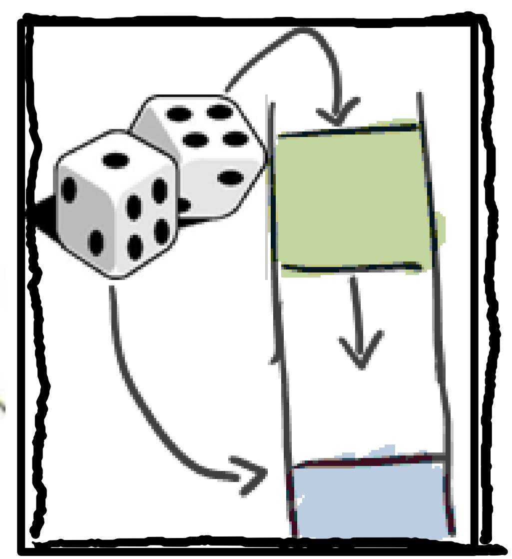
The most popular language in the world



http://www.langpop.com/

Defensive measures

- NX bit / DEP / W⊕X
- Canaries and Cookies
- ASLR



Not

Good

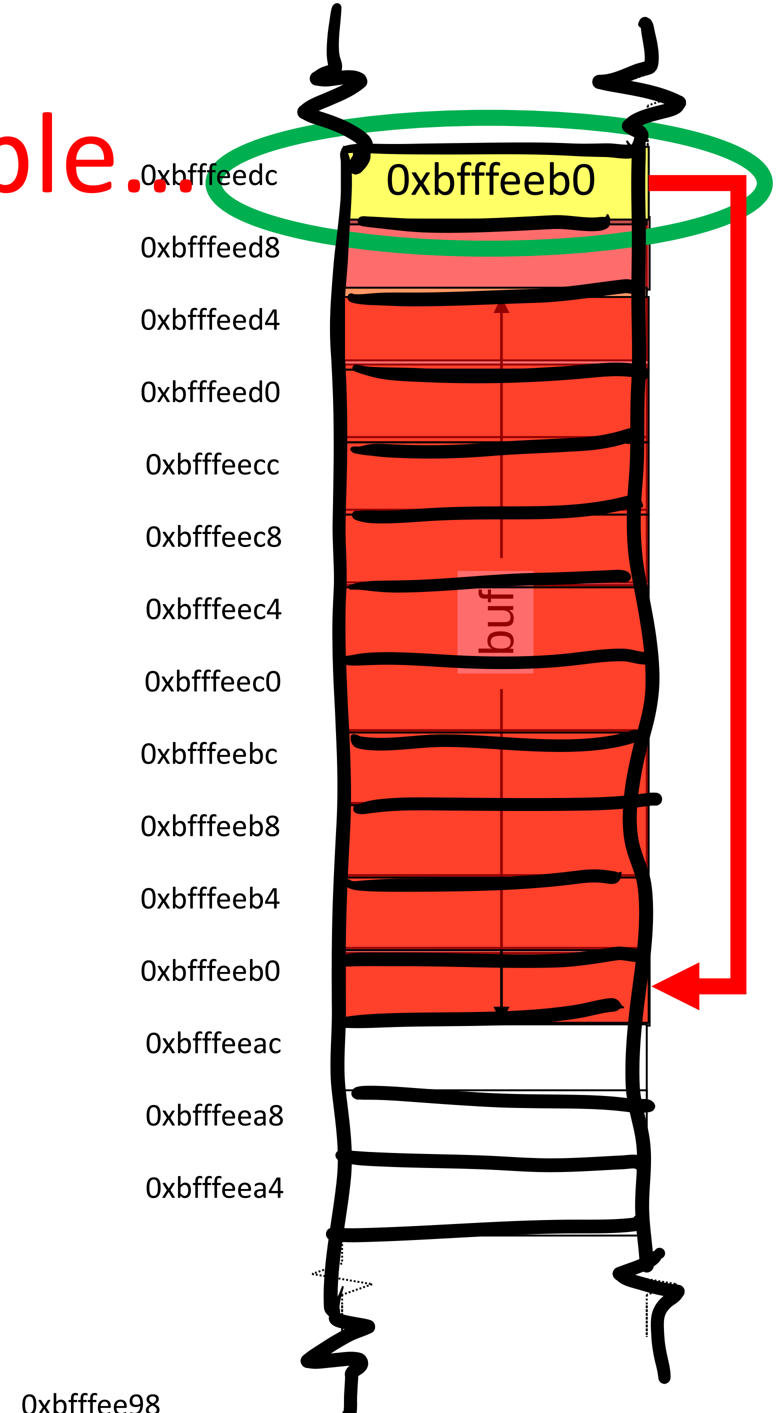
Enough

Why not?

Let us have a look at modern exploitation

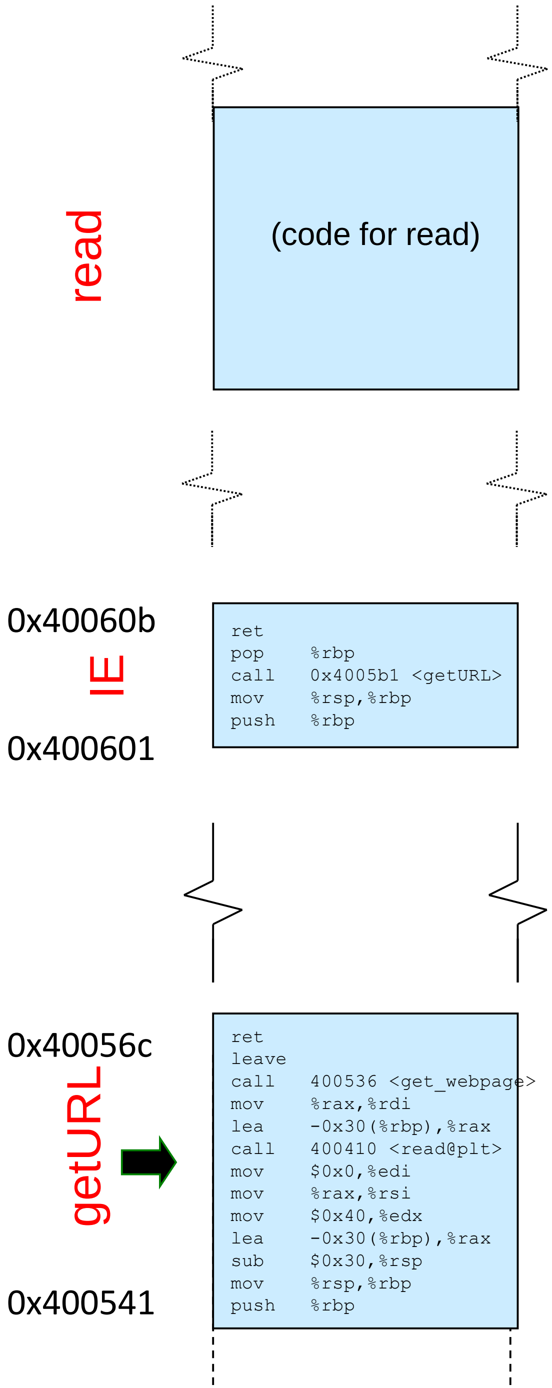
It used to be so simple…

It used to be so simple…



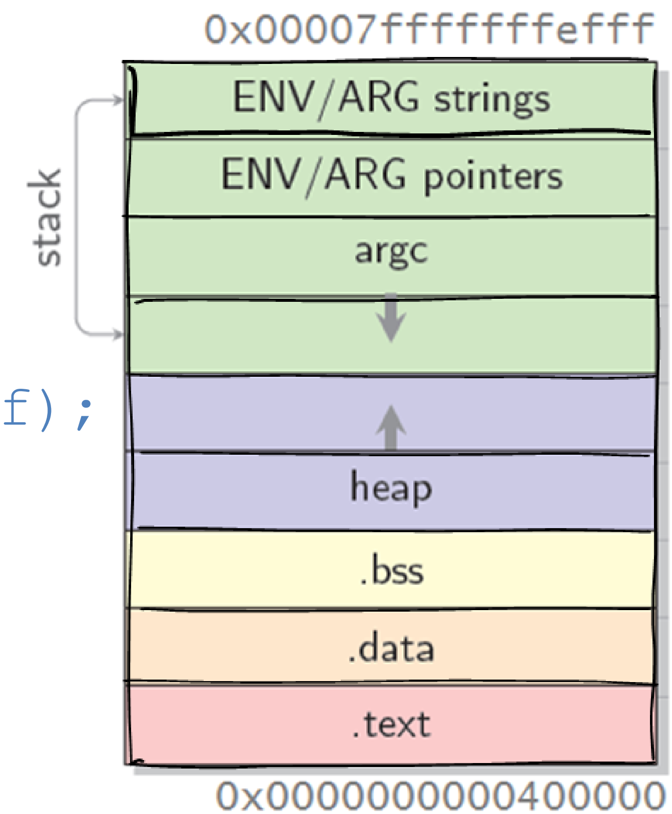
A simple example...

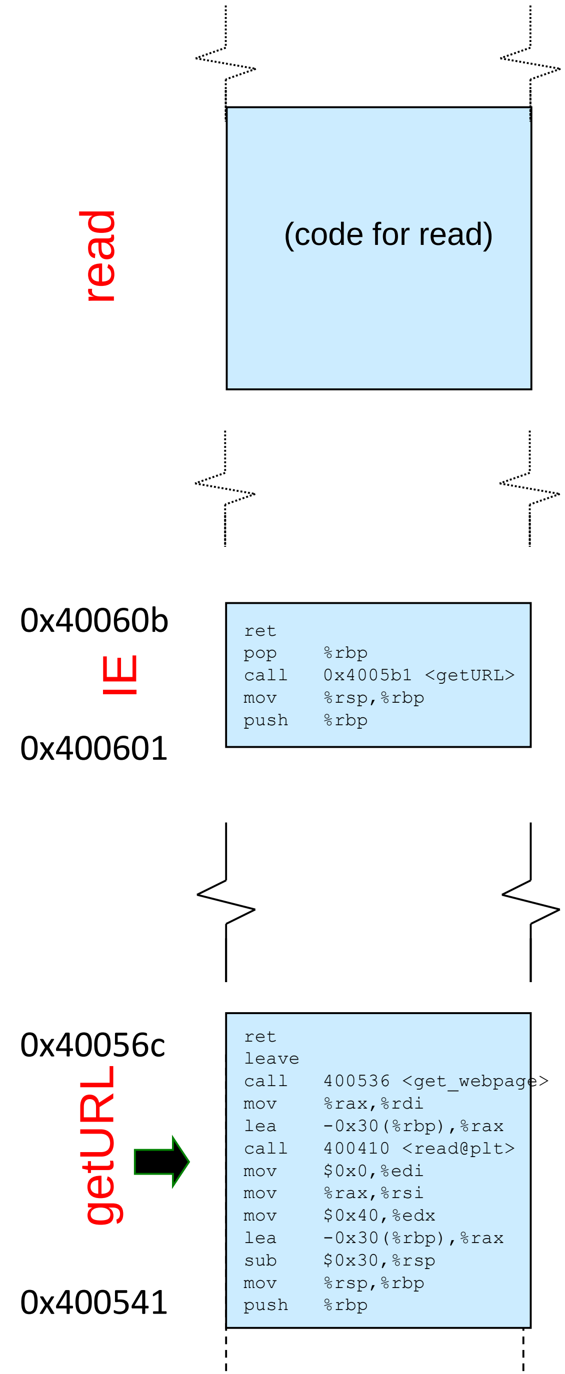
```
getURL ()
{
   char buf[48];
   read(0,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



Memory layout of a process...

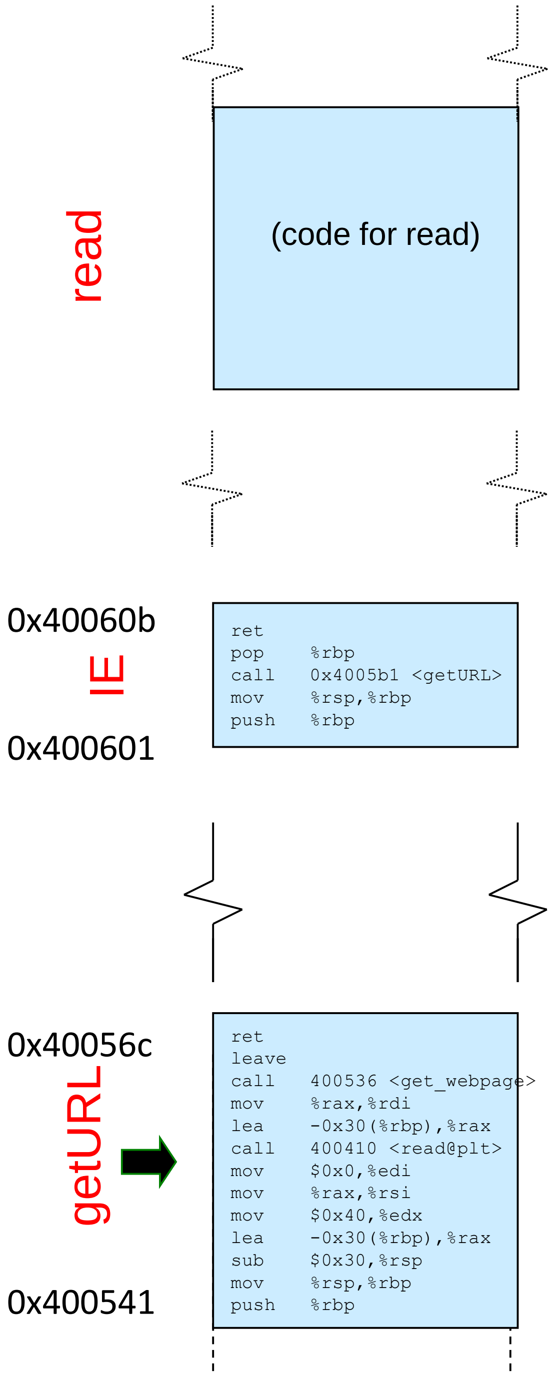
```
getURL ()
{
  char buf[48];
  read(0,buf,64);
  get_webpage (buf);
}
IE ()
{
  getURL ();
}
```





Memory layout of a process...

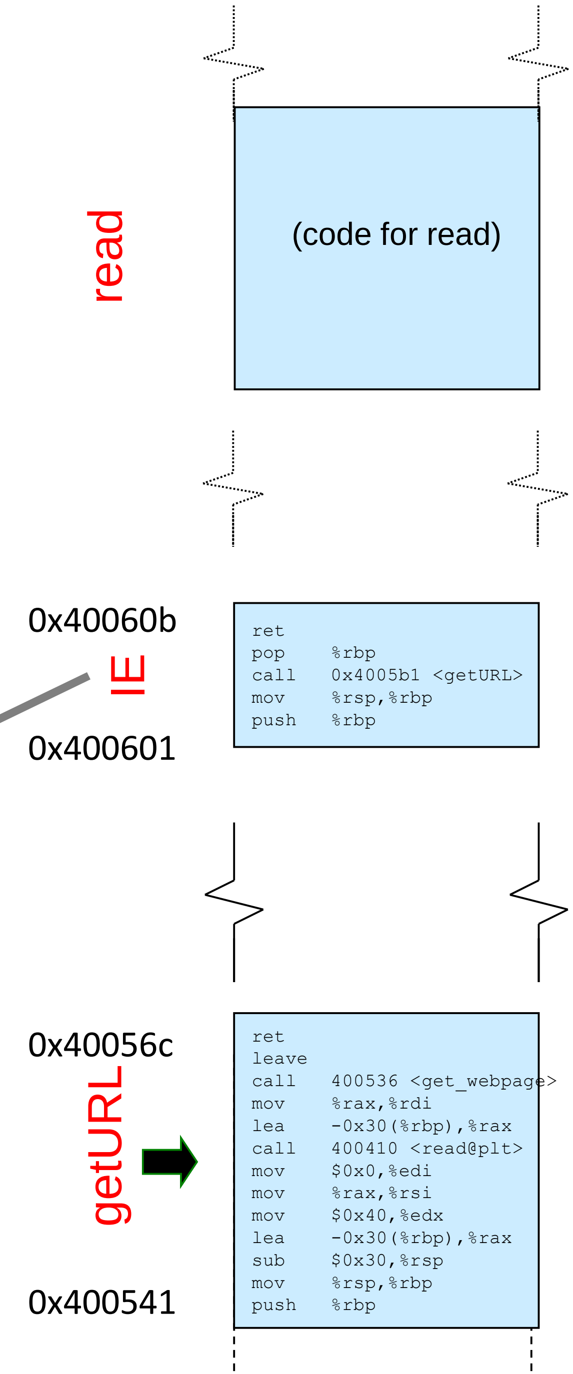
```
getURL ()
{
   char buf[48];
   read(0,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



Memory layout of a process...

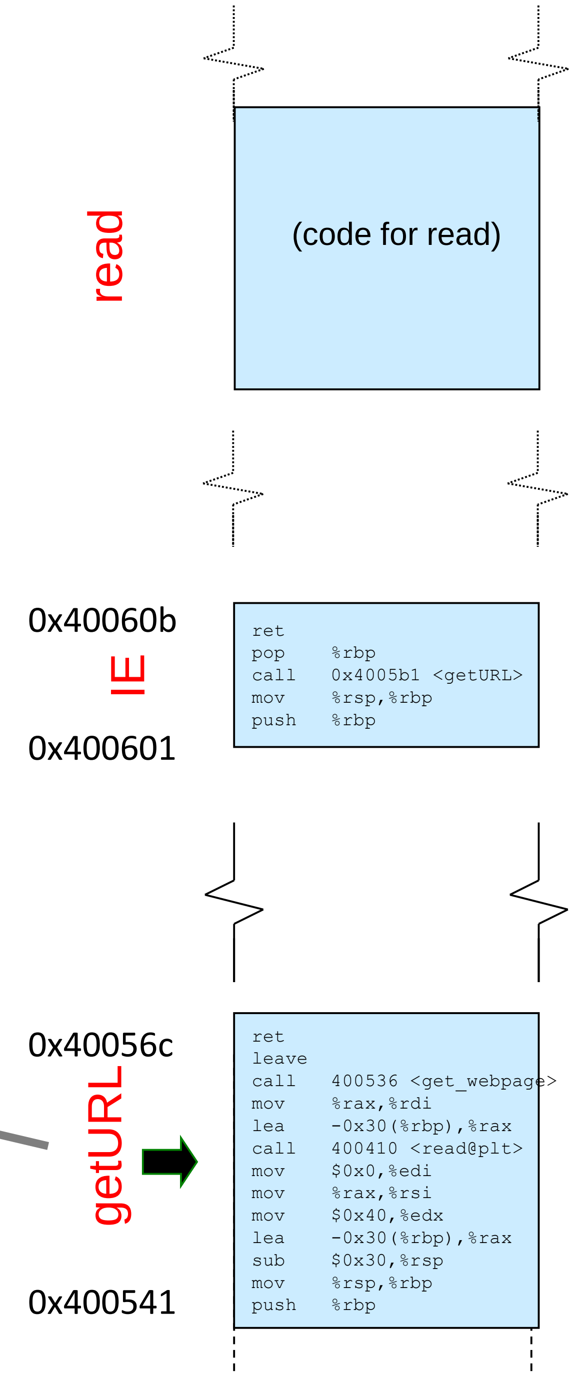
```
getURL ()
{
   char buf[48];
   read(0,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```

```
0x400601: push    %rbp
0x400602: mov     %rsp,%rbp
0x400605: call    0x4005b1 <getURL>
0x40060a: pop     %rbp
0x40060b: ret
```



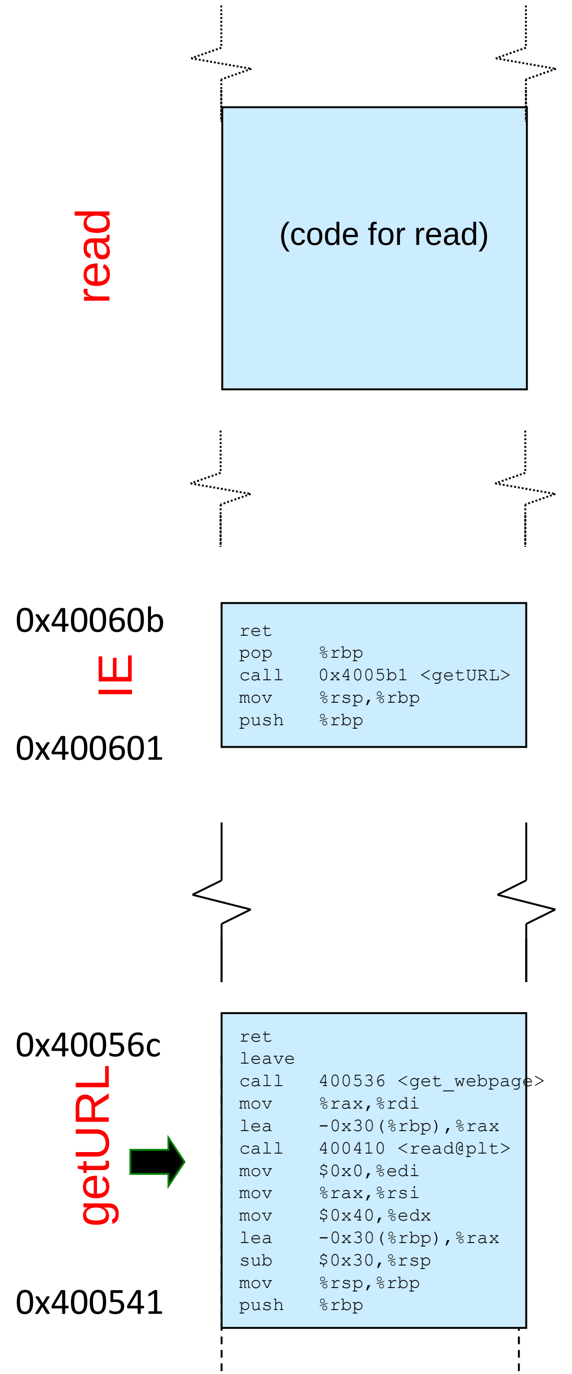
Memory layout of a process…

```
400541: push   %rbp
400542: mov    %rsp,%rbp
400545: sub    $0x30,%rsp
400549: lea    -0x30(%rbp),%rax
40054d: mov    $0x40,%edx
400552: mov    %rax,%rsi
400555: mov    $0x0,%edi
40055a: call   400410 <read@plt>
40055f: lea    -0x30(%rbp),%rax
400563: mov    %rax,%rdi
400566: call   400536 <get_webpage>
40056b: leave
40056c: ret
```



Memory layout of a process...

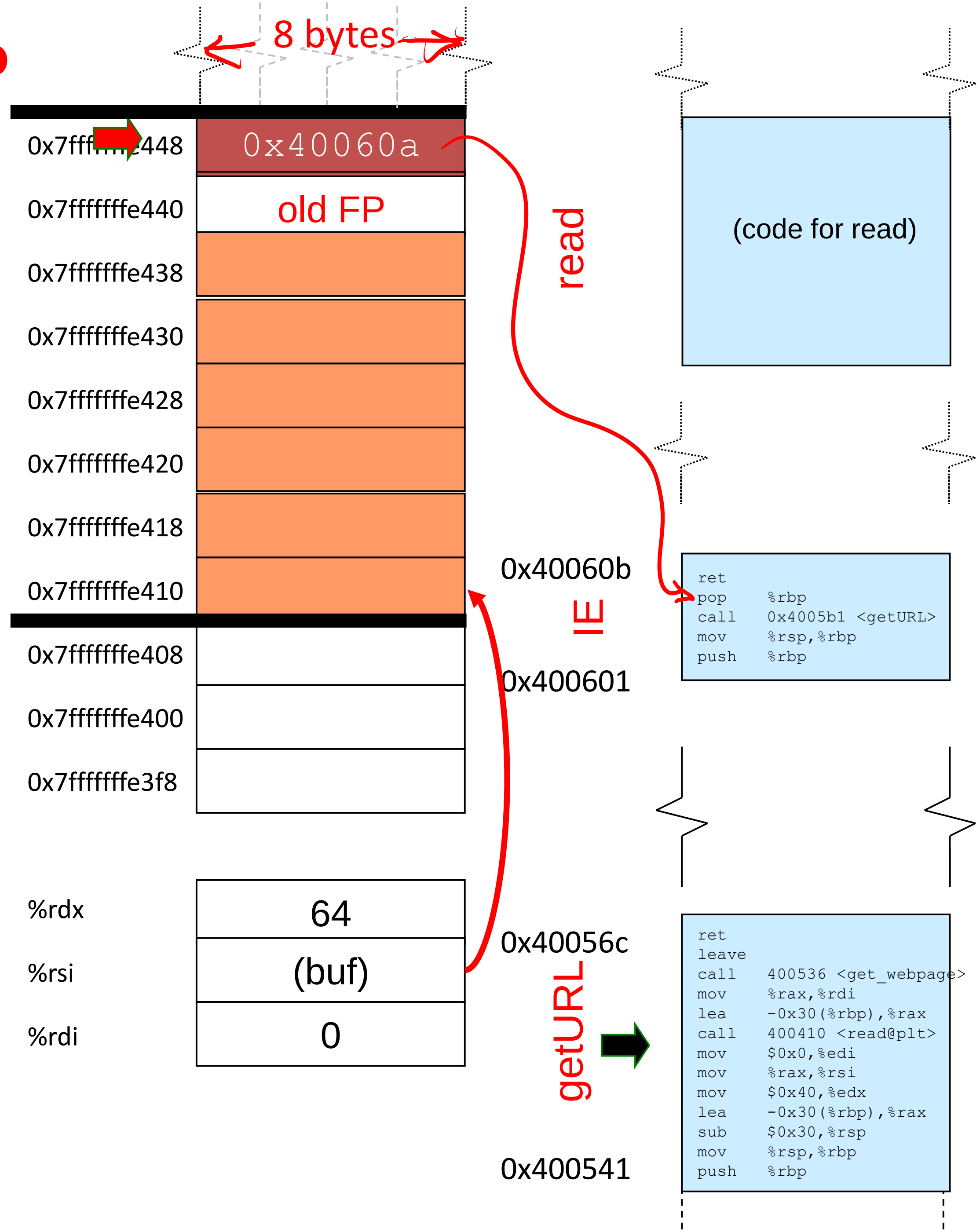
```
getURL ()
{
   char buf[48];
   read(0,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



What about stack?

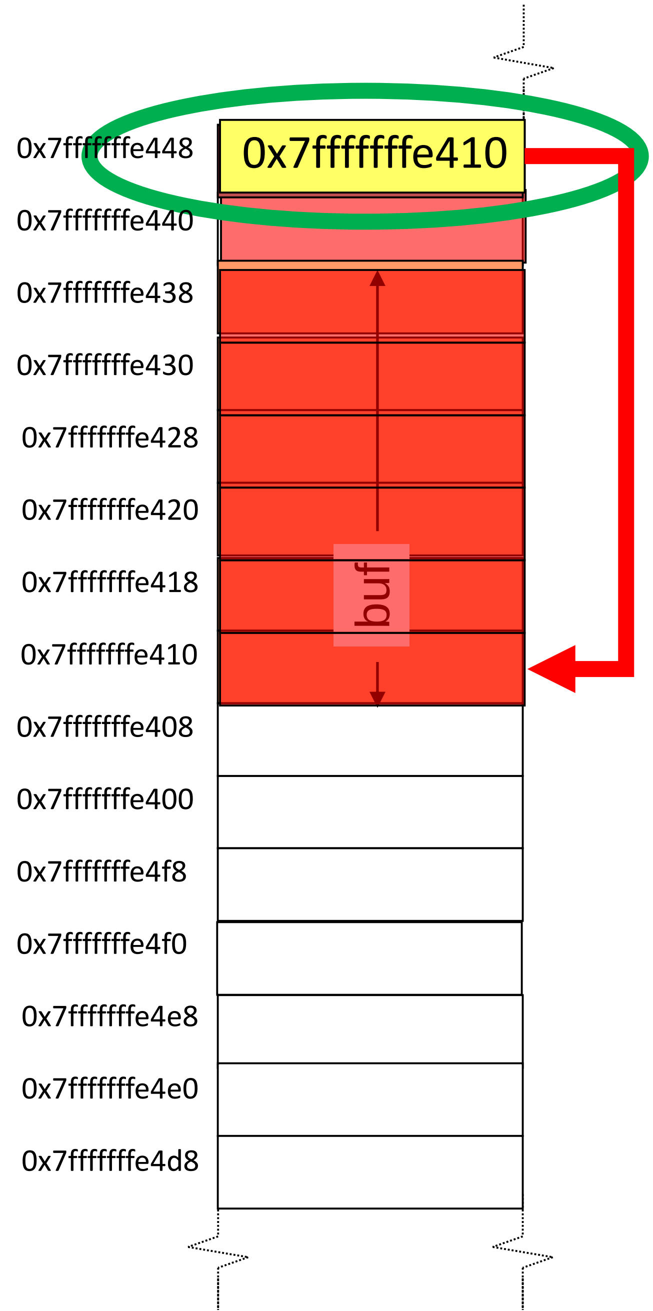
When getURL is about to call 'read'

```
getURL ()
{
   char buf[48];
   read(0,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



Exploit

```
getURL ()
{
   char buf[48];
   read(fd, buf, 64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



0xbfffee98

That is it, really

- Only need to stick a program in the buffer

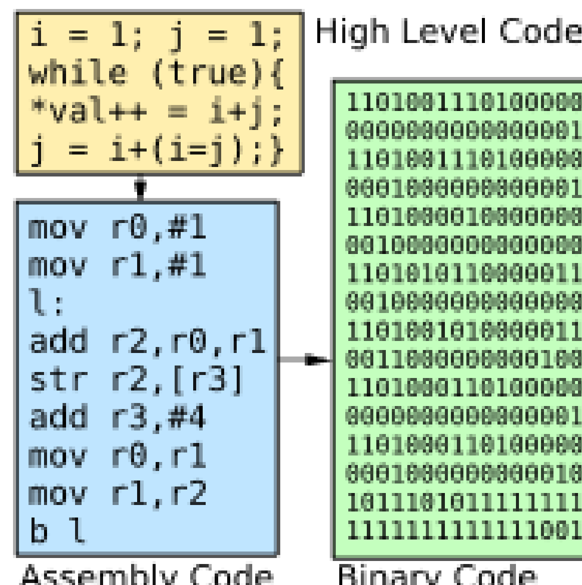
  Easy to do: attacker controls what goes in the buffer!

  – and that program simply consists of a few instructions (not unlike what we saw before)

Two phases

1. Divert the control flow

2. Execute code



That is, fundamentally, it.

- Let us see whether we understood this.

Can you exploit this?

```
char gWelcome [] = "Welcome to our system! ";

void echo (int fd)
{
  int len;
  char name [64], reply [128];

  len = strlen (gWelcome);
  memcpy (reply, gWelcome, len); /* copy the welcome string to reply */

  write_to_socket (fd, "Type your name: "); /* prompt client for name */
  read (fd, name, 128);                     /* read name from socket */

  /* copy the name into the reply buffer (starting at offset len, so
   * that we won't overwrite the welcome message we copied earlier). */
  memcpy (reply+len, name, 64);

  write (fd, reply, len + 64); /* now send full welcome message to client */
  return;
}

void server (int socketfd) { /* just call echo() in an endless loop */
  while (1)
    echo (socketfd);
}
```

Can you exploit this?

without comments

```
char gWelcome [] = "Welcome to our system! ";

void echo (int fd)
{
  int len;
  char name [64], reply [128];

  len = strlen (gWelcome);
  memcpy (reply, gWelcome, len);

  write_to_socket (fd, "Type your name: ");
  read (fd, name, 128);



  memcpy (reply+len, name, 64);

  write (fd, reply, len + 64);
  return;
}

void server (int socketfd) {
  while (1)
    echo (socketfd);
}
```

HOW DO WE STOP THE ATTACKS?

- The best defense is proper bounds checking
- but there are many C/C++ programmers and some are bound to forget

→ Are there any *system* defenses that can help?

So far

CLASSIC ATTACKS

Now

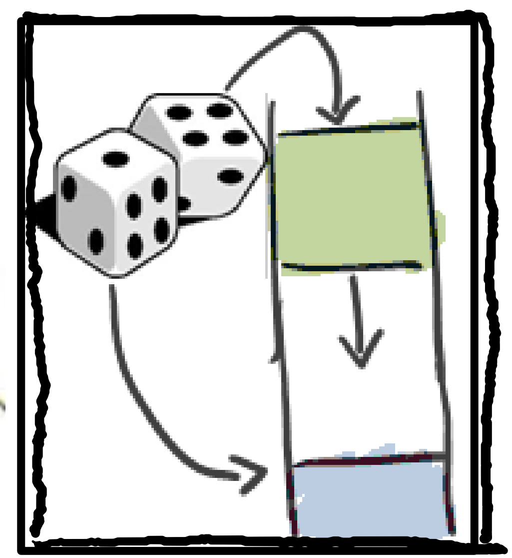
CLASSIC DEFENSES

and

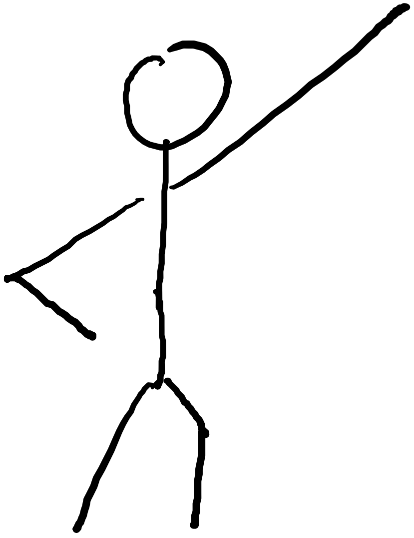
NEO-CLASSIC ATTACKS AGAINST CLASSIC DEFENSES

No more

Defenses: stack protection, DEP, ASLR



Stack Canaries!



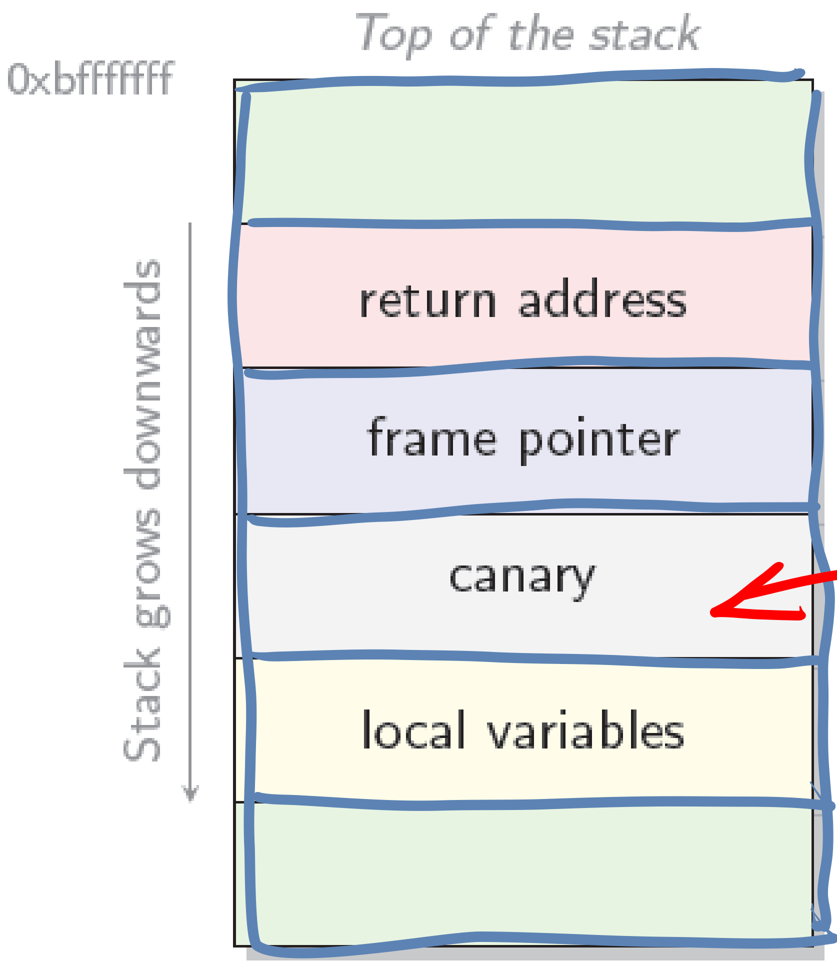
Threat Model

- Attackers are smashing the stack
- Using contiguous overflow
- Divert control to injected code

Compiler-level techniques
Canaries

- Goal: make sure we detect overflow of return address
  - The functions' prologues insert a *canary* on the stack
  - The canary is a random 64-bit value inserted between the return address and local variables
- The epilogue checks if the canary has been altered
- Drawback: requires recompilation

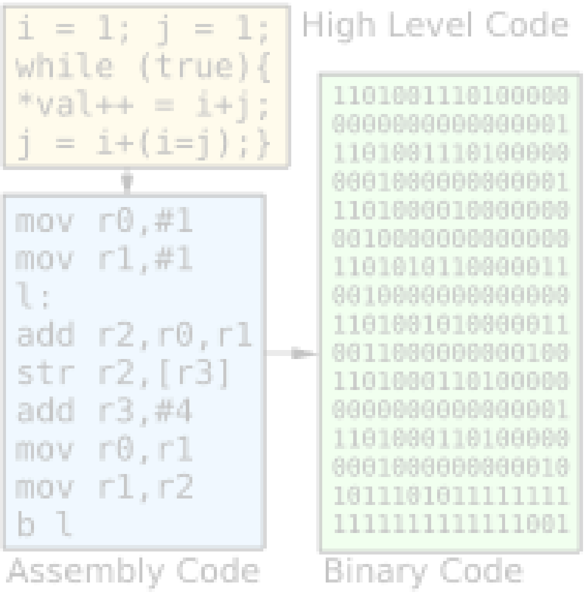
Canaries


Top of the stack
0xbfffffff
Stack grows downwards
return address
frame pointer
canary
local variables

Recall: two phases

1. Divert the control flow

2. Execute code



How good are they?

- Assume random canaries protect the stack

Can you still exploit this?

```
char gWelcome [] = "Welcome to our system! ";

void echo (int fd)
{
  int len;
  char name [64], reply [128];

  len = strlen (gWelcome);
  memcpy (reply, gWelcome, len);

  write_to_socket (fd, "Type your name: ");
  read (fd, name, 128);



  memcpy (reply+len, name, 64);

  write (fd, reply, len + 64);
  return;
}

void server (int socketfd) {
  while (1)
    echo (socketfd);
}
```

Threat Model

- Attackers are smashing the stack
- Using contiguous overflow
- Divert control to injected code

“DEP!”

DEP / NX bit / W⊕X

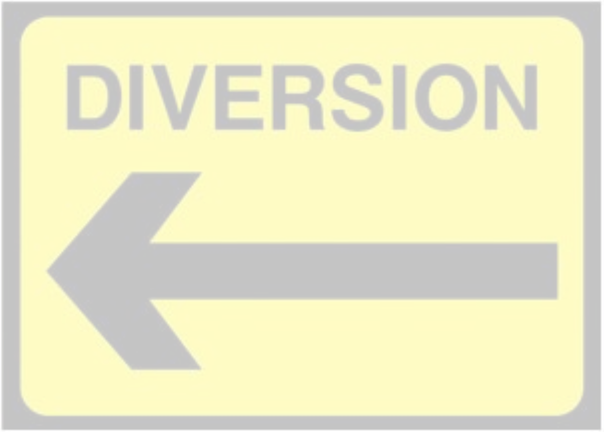
- Idea: separate executable memory locations from writable ones
  - A memory page cannot be both writable and executable at the same time
- "Data Execution Prevention (DEP)"

Threat Model

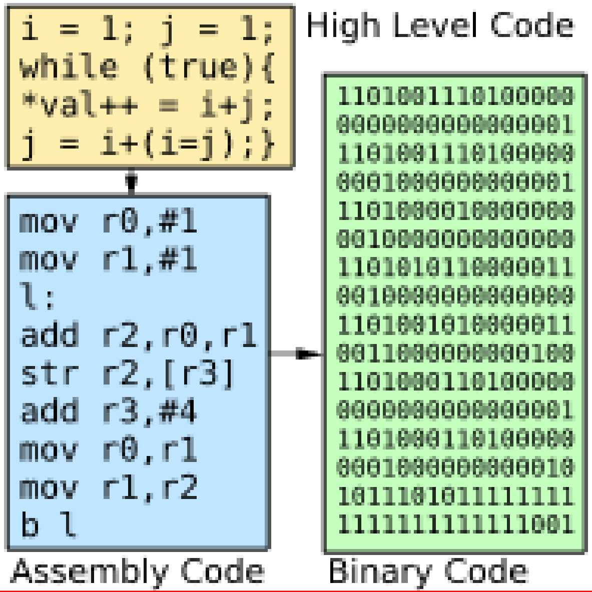
Attackers divert control to injected code

Recall: two phases

1. Divert the control flow



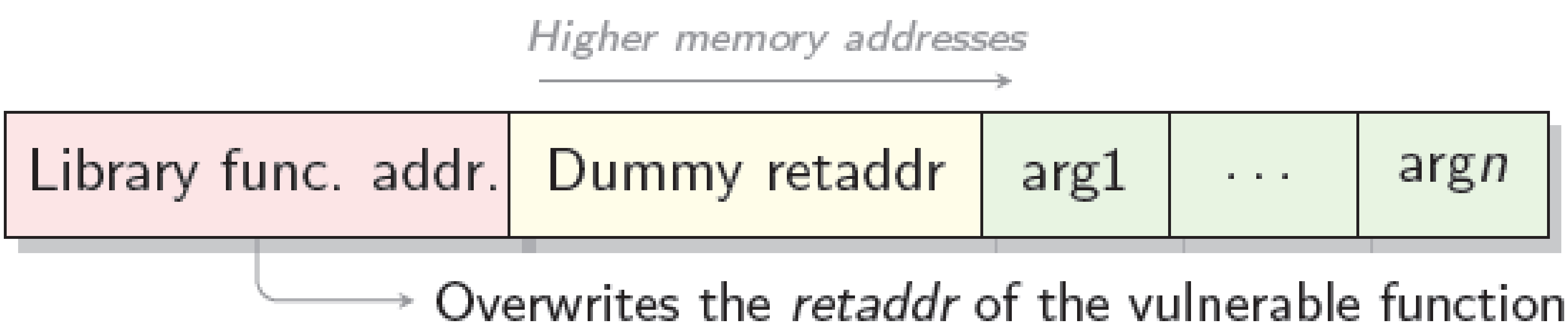
2. Execute code



So how to get a shell now?

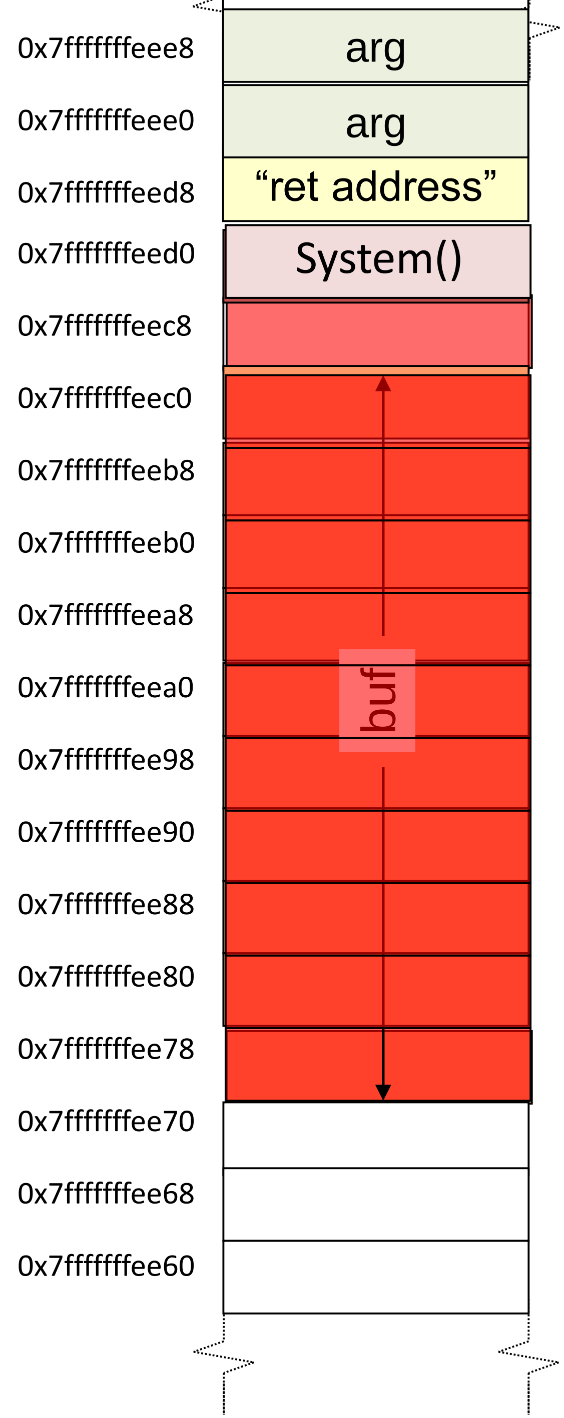
Bypassing DEP (32 bit)

- Return into libc
- Three assumptions:
  - We can manipulate a code pointer
  - The stack is writable
  - We know the address of a “suitable" library function (e.g., system())

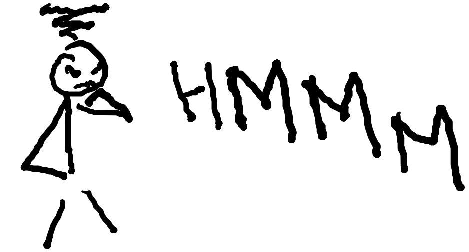


Stack

- Why the "ret address"?
- What could we do with it?



Bypassing DEP (64 bit)



- 64-bit: parameters passed in registers rather than stack
  - So we need to be able to control the %rdi, %rsi, %rdx, … registers to make return into libc work
  - Whether this is possible depends on register allocation, which depends on compiler (settings)

➔ HARD!

So…

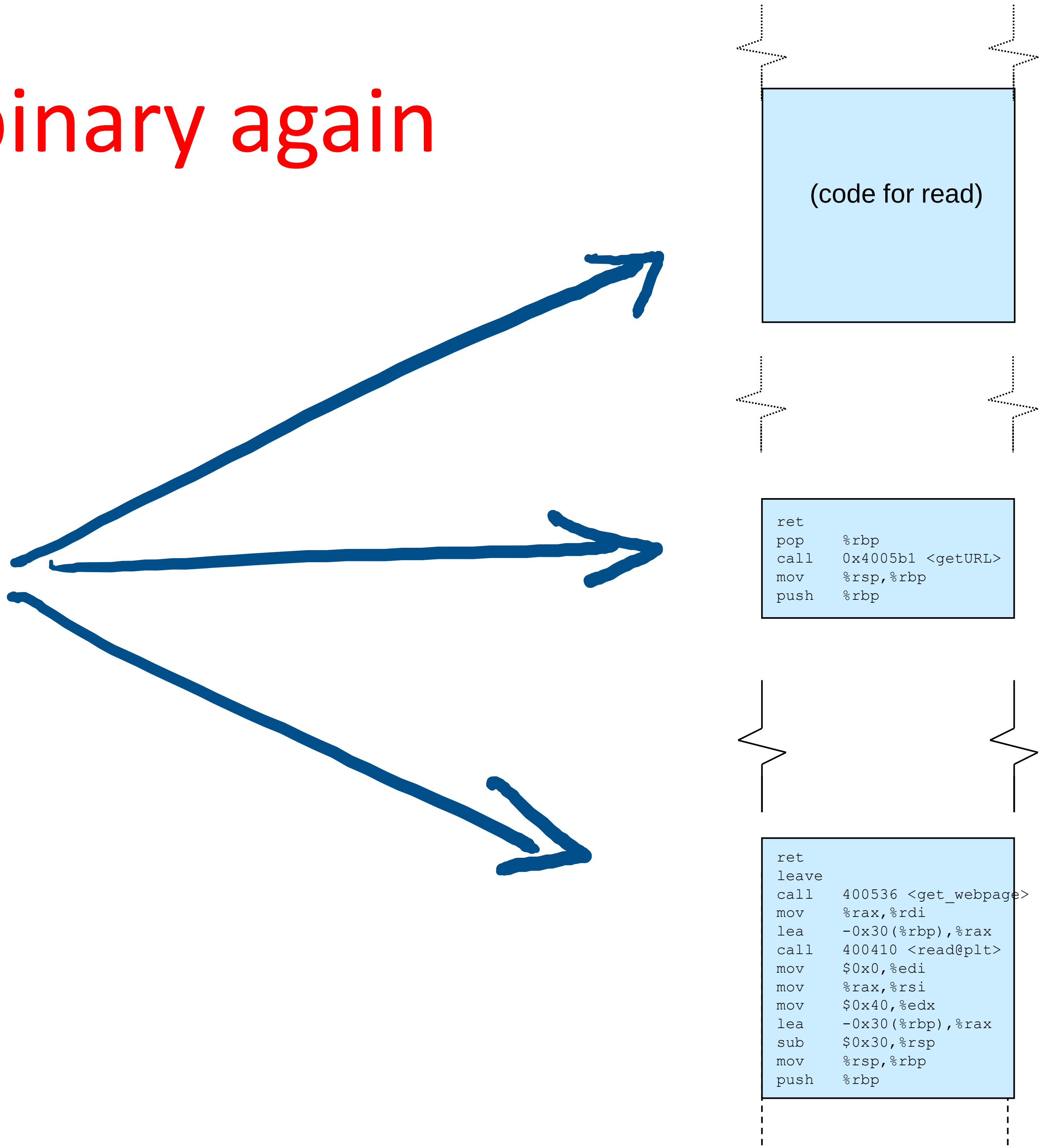
more

extreme

code reuse

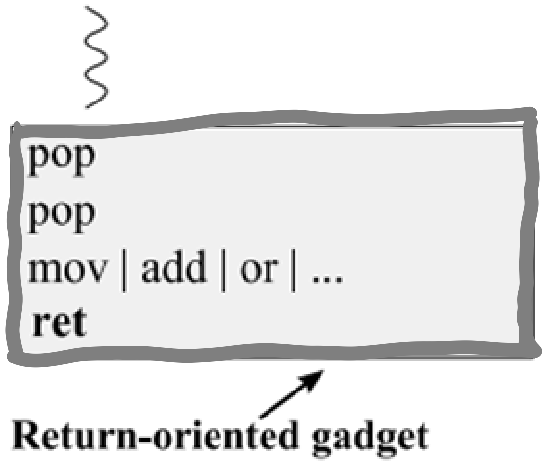
Consider binary again

Lots of code!



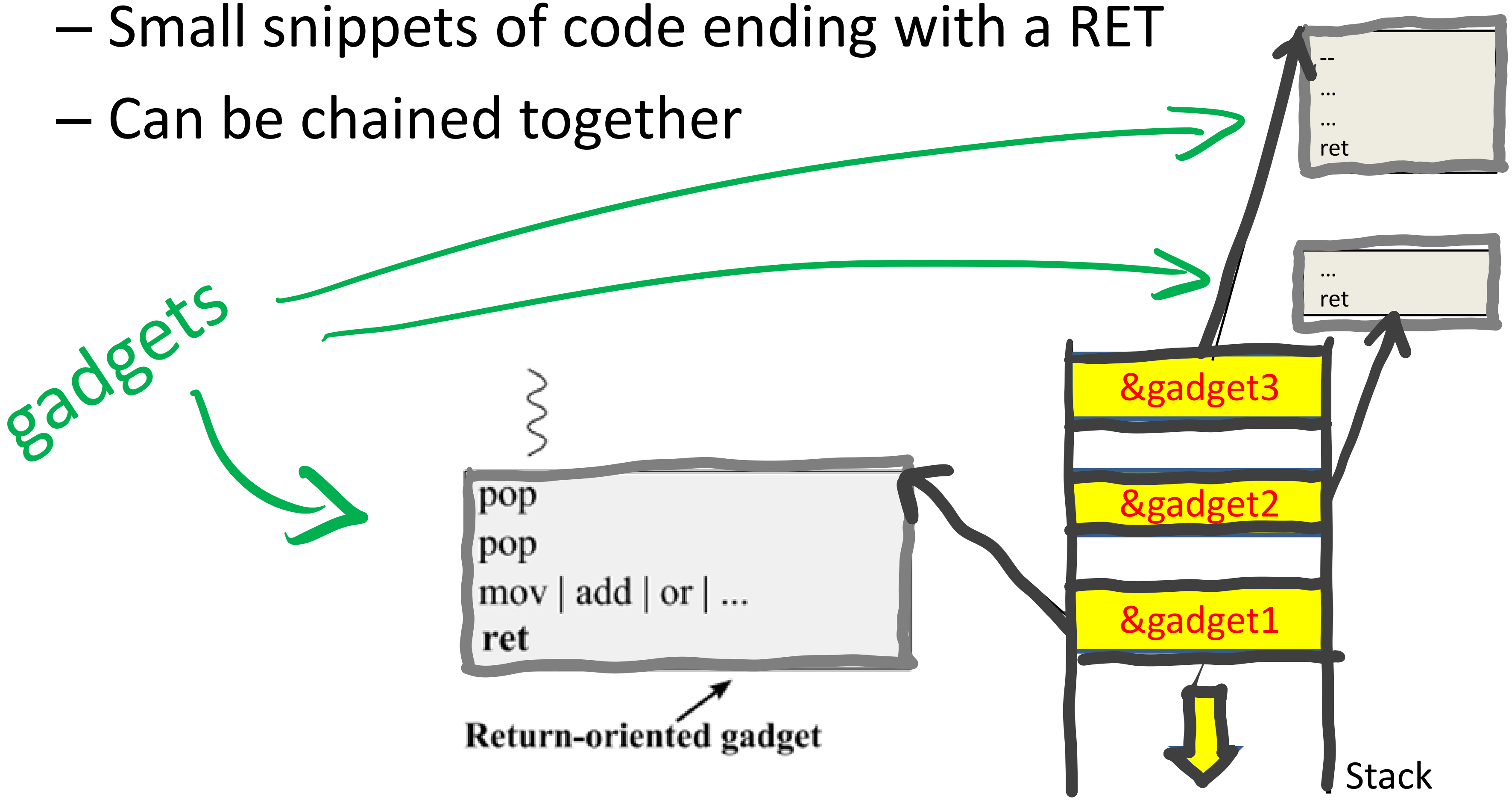
Return Oriented Programming (ROP)

- Small snippets of code ending with a RET
- Can be chained together



ROP

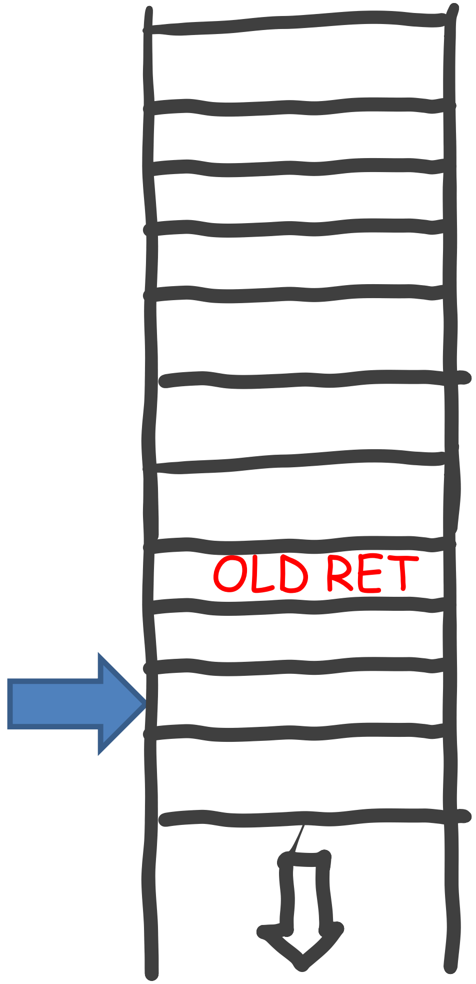
- Small snippets of code ending with a RET
- Can be chained together



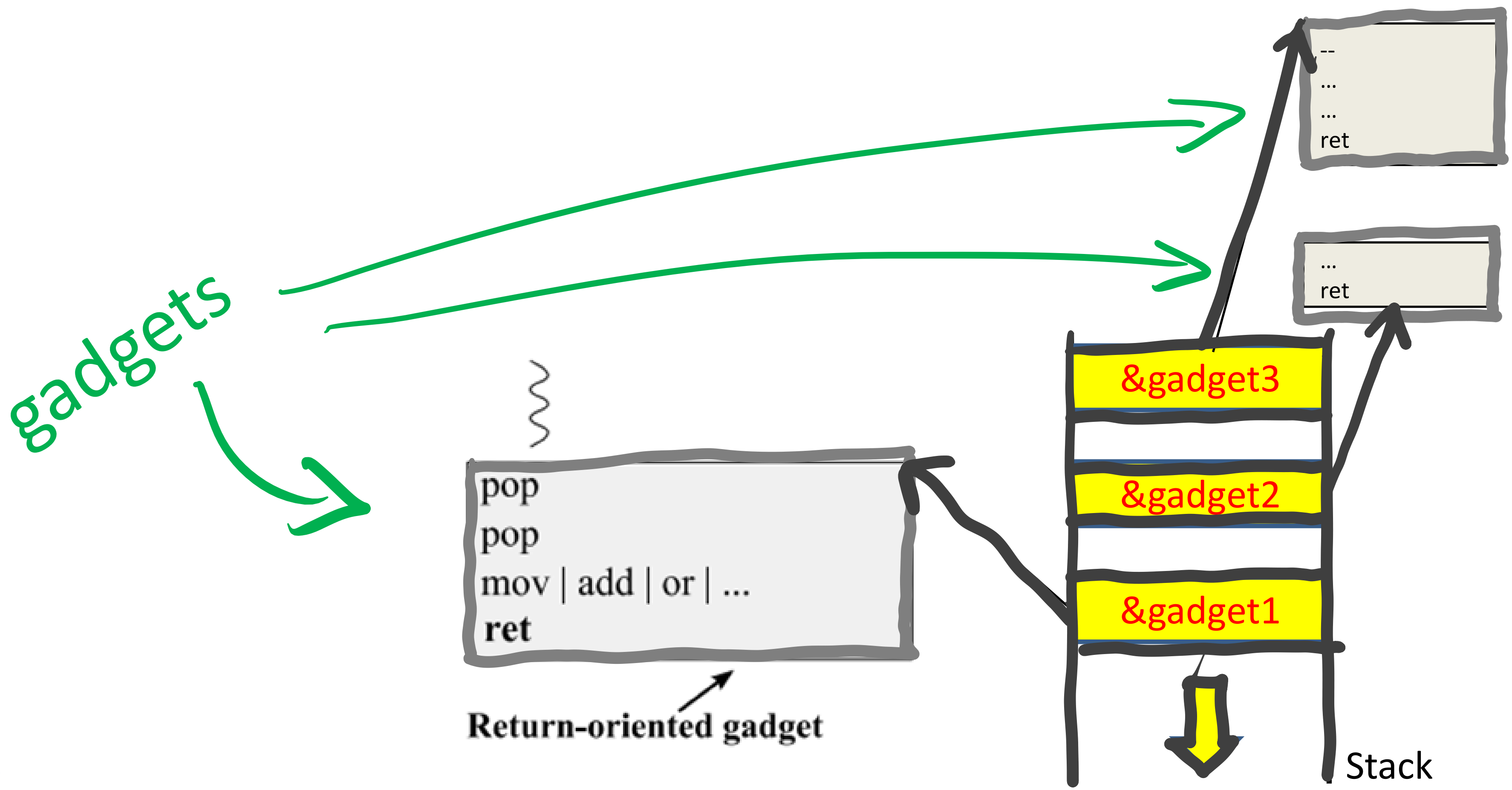
Example: want to call exit with code 5

(exit syscall number = 60, return value in %rdi)

```
0x400536     POP %rbx
             POP %rax
             ADD %rbx, %rax
             RET
      .
      .
      .
0x4000640    MOV %rbx,%rdi
             RET
      .
      .
      .
0x4000768    INT $0x80
```



ROP



VU University Amsterdam

How good are they?

- Assume random canaries protect the stack
- Assume DEP prevents execution of the stack

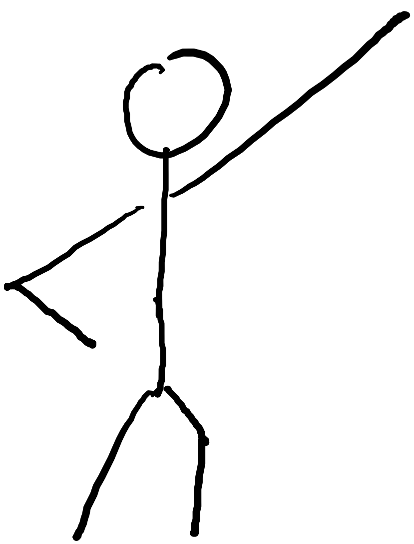
Can you still exploit this?

```
char gWelcome [] = "Welcome to our system! ";

void echo (int fd)
{
  int len;
  char name [64], reply [128];

  len = strlen (gWelcome);
  memcpy (reply, gWelcome, len);

  write_to_socket (fd, "Type your name: ");
  read (fd, name, 128);



  memcpy (reply+len, name, 64);

  write (fd, reply, len + 64);
  return;
}

void server (int socketfd) {
  while (1)
    echo (socketfd);
}
```

Threat Model

Attackers divert control to injected code

ASLR!



Let us make it a little harder still…

Address Space Layout Randomisation

- Idea:
  - Re-arrange the position of key data areas randomly (stack, .data, .text, shared libraries, . . .)
  - Buffer overflow: the attacker does not know the address of the shellcode
  - Return-into-libc: the attacker can't predict the address of the library function
  - Implementations:
    - Linux kernel > 2.6.11,
    - Windows >= Vista, . . .

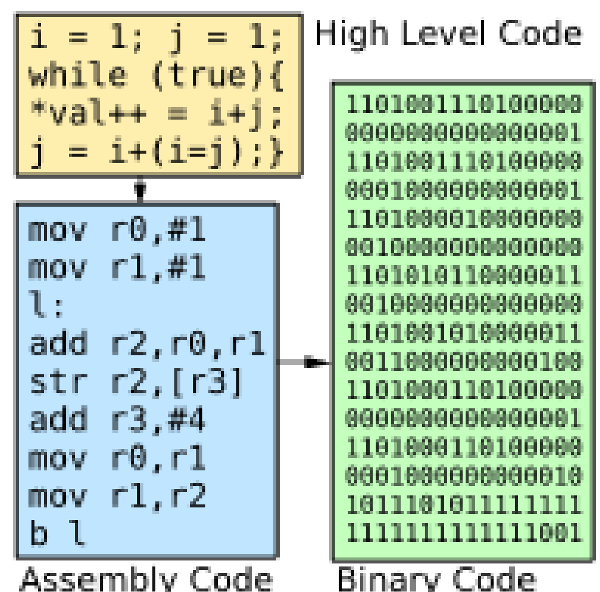
Threat Model

To divert the control, attackers need to know in advance the location of that code

Recall: two phases

1. Divert the control flow

2. Execute code



So how to get a shell now?

ASLR: Problems

- 32-bit implementations use few randomisation bits
- An attacker can still exploit non-randomised areas, or
- rely on other information leaks (e.g., format bug)
  - ➔ typically known as "memory disclosures"

- So… (I bet you saw this one coming)….

How good are they?

- Assume random canaries protect the stack
- Assume DEP prevents execution of the stack
- Assume ASLR randomized the stack *and* the start address of the code
  - but let us assume that all functions are still at the same relative offset from start address of code
  - (in other words: need only a single code pointer)

Can you still exploit this?

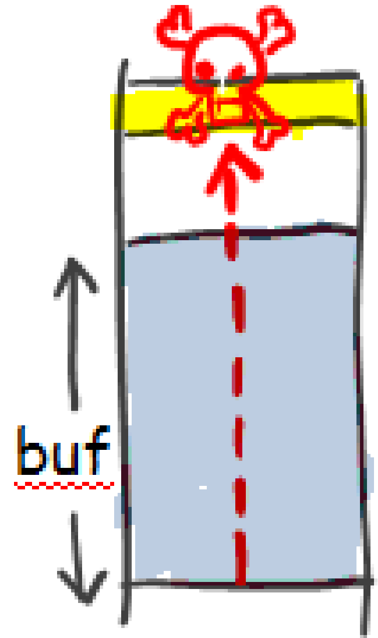
```
char gWelcome [] = "Welcome to our system! ";

void echo (int fd)
{
  int len;
  char name [64], reply [128];

  len = strlen (gWelcome);
  memcpy (reply, gWelcome, len);

  write_to_socket (fd, "Type your name: ");
  read (fd, name, 128);



  memcpy (reply+len, name, 64);

  write (fd, reply, len + 64);
  return;
}

void server (int socketfd) {
  while (1)
    echo (socketfd);
}
```

Threat Model

To divert the control, attackers need to know ~~in advance~~ the location of that code

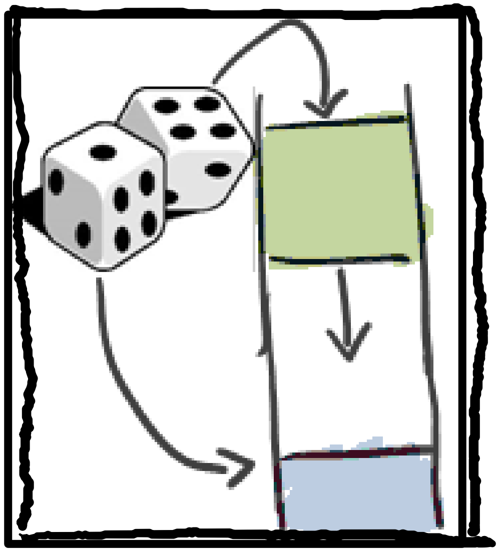
Incidentally, so far we only talked about buffer overflows

- Perpetual top-3 threat
  - SANS CWE Top 25 Most dangerous programming errors
- But not the only one
  - Temporal errors, integer overflows, format strings…



So far: main defensive measures

- NX bit / DEP / W⊕X
- Canaries and Cookies
- ASLR



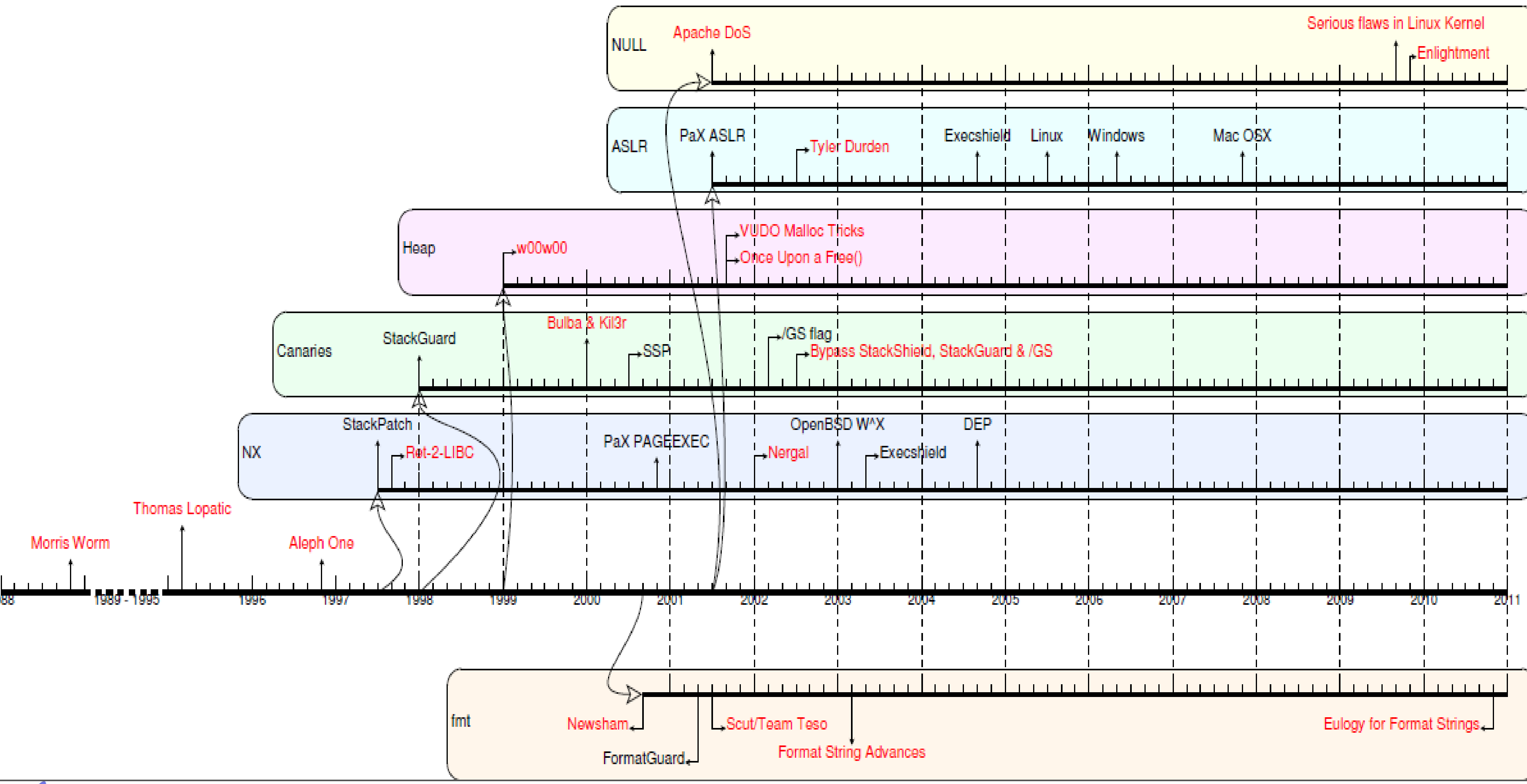
Not

Good

Enough

Evolution at work

“Memory Errors: the Past, the Present and the Future” [RAID’12]



VU University Amsterdam

Let us

Take a Step Back

What is the attacker’s game?

Two *fundamental* requirements:

- locate code (gadgets)
- jump to it

All Defenses

- Try to restrict the attacker in some way
  - Stop contiguous overflow
  - Stop code injection
  - Stop ability to target code
  - Stop …whatevever…

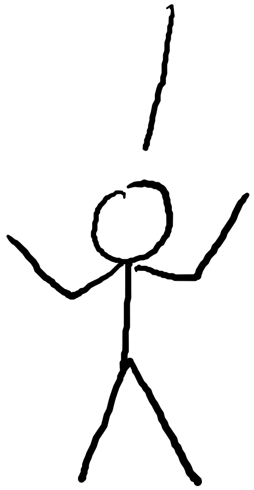
- Crucial question always: “What is left?”

All Defenses

- Have a threat model in mind
  - Attacker overwrites return address
  - Attacker overwrites return address or function pointer
  - Attacker overwrites return address or function pointer or data
  - Attacker cannot leak randomization
  - Attacker cannot crash a system
  - …
- Crucial question : "Is it realistic and what is left?"
  Crucial answer : "We always get it wrong!"

Defenses can be bypassed

- Easily or with difficulty
- So, useless?

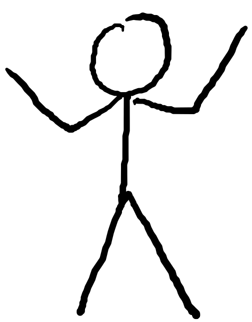


Defenses can be bypassed

- Easily or with difficulty
- So, useless?
- **No! They still make the attackers' life difficult**

IT'S THE RED QUEEN EFFECT

ALICE AND THE RED QUEEN ARE RUNNING AS FAST AS THEY CAN SO THEY STAND STILL



JUST LIKE LIONS AND ZEBRA'S BOTH KEPT EVOLVING BUT NEITHER GOT ANY BETTER. THE ZEBRA RUNS FASTER BUT SO DOES THE LION

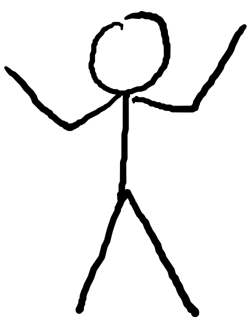
IT'S THE **RED QUEEN** EFFECT

/ NOBODY WINS

THEY ARE RUNNING TO STAND STILL

POINTLESS!

ALICE AND THE RED QUEEN ARE RUNNING AS FAST AS THEY CAN SO THEY STAND STILL



JUST LIKE LIONS AND ZEBRA'S BOTH KEPT EVOLVING BUT NEITHER GOT ANY BETTER. THE ZEBRA RUNS FASTER BUT SO DOES THE LION

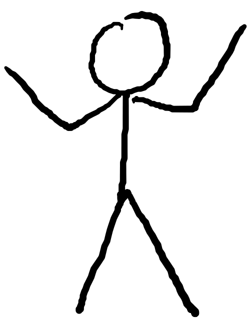
IT'S THE RED QUEEN EFFECT

/ NOBODY WINS

THEY ARE RUNNING TO STAND STILL

POINTLESS!

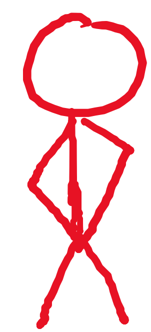
ALICE AND THE RED QUEEN ARE RUNNING AS FAST AS THEY CAN SO THEY STAND STILL



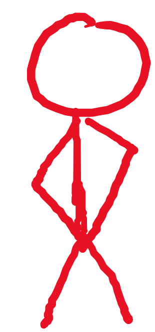
JUST LIKE LIONS AND ZEBRA'S BOTH KEPT EVOLVING BUT NEITHER GOT ANY BETTER. THE ZEBRA RUNS FASTER BUT SO DOES THE LION

REALLY?

JUST ASK THE ZEBRA WHAT HAPPENS IF IT STOPPED RUNNING!



ALSO THE DEFENSES DO MAKE LIFE MORE DIFFICULT FOR THE ATTACKER



We have seen this already

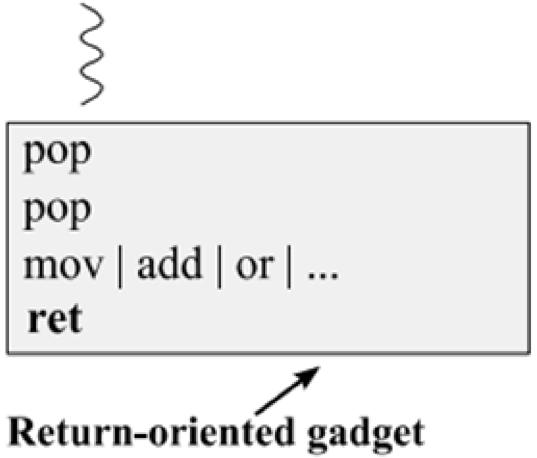
Let us explore this further

Consider

ROP

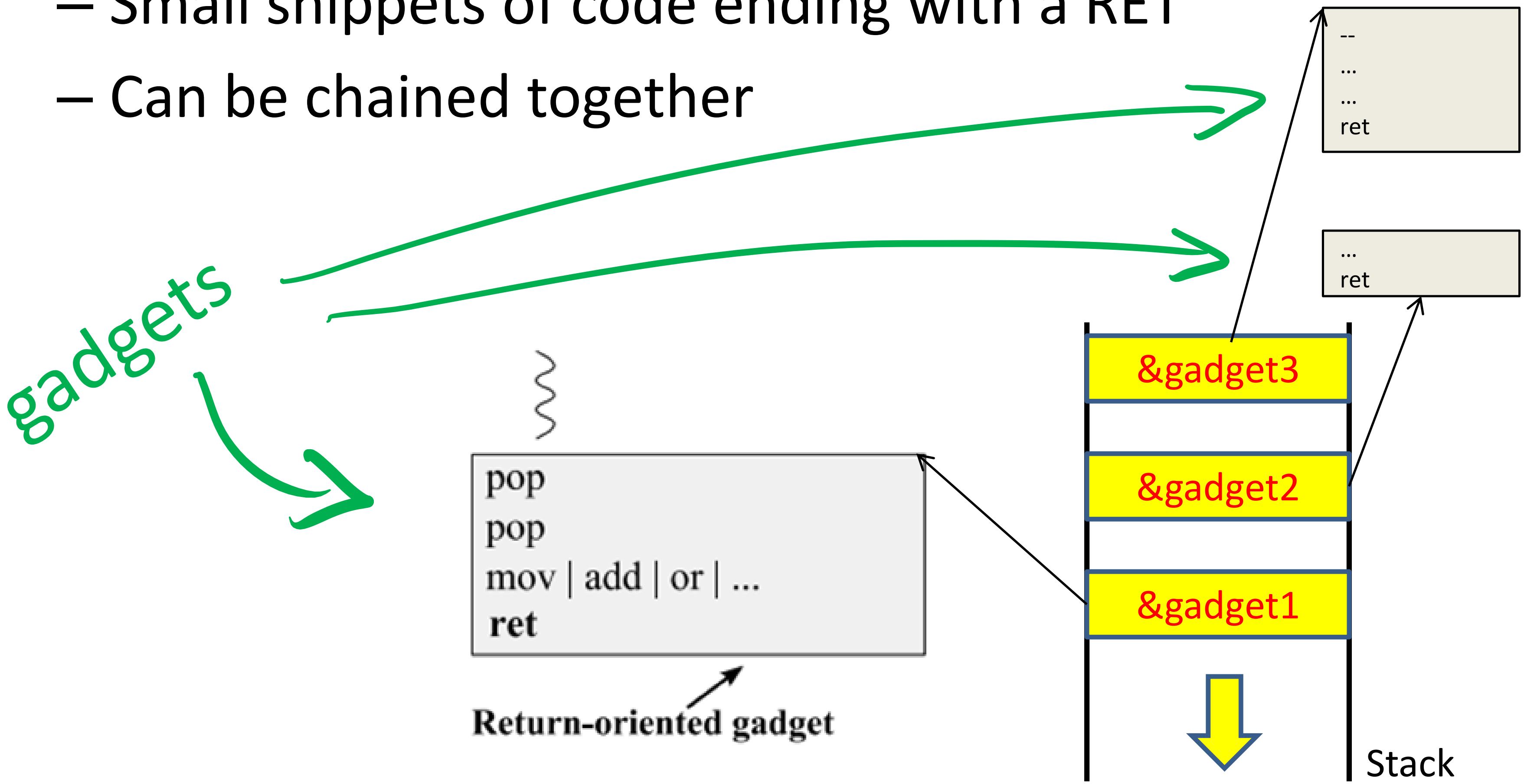
ROP

– Small snippets of code ending with a RET
– Can be chained together

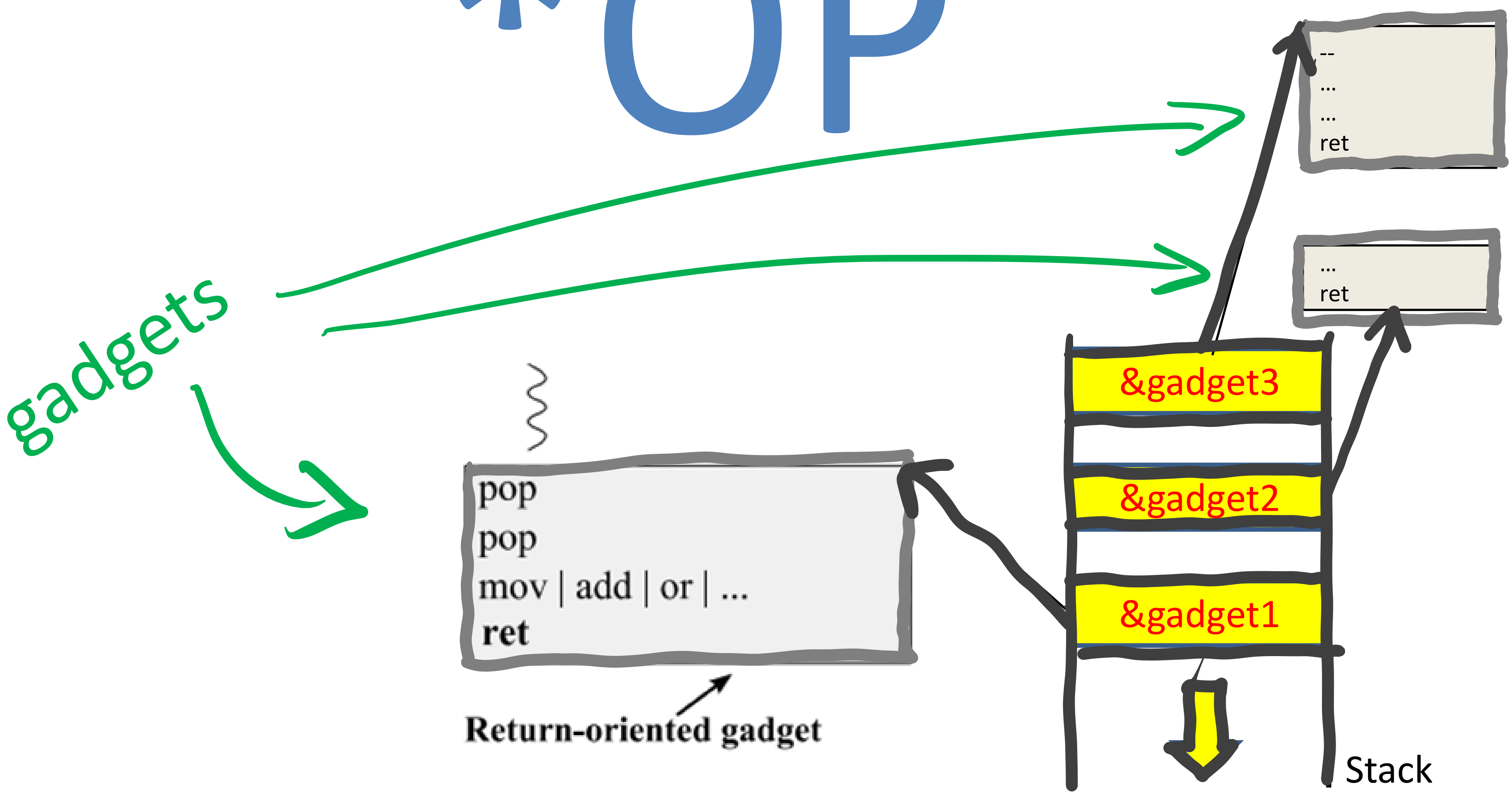


ROP

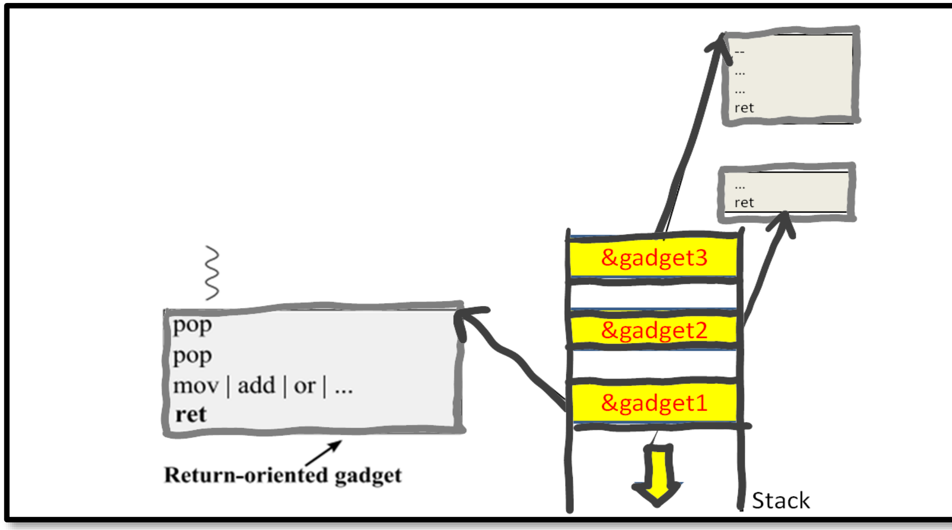
– Small snippets of code ending with a RET
– Can be chained together



JOP ROP COP
*OP



ROP is not easy!



- Need to know addresses of gadgets in detail
- Need exact same version of the binary

→ NOT Portable

SROP

SigReturn Oriented Programming

Erik Bosman

Portable!

Need as few as one gadget:

"**syscall** (0x0f05) & **ret** (0xc3)"

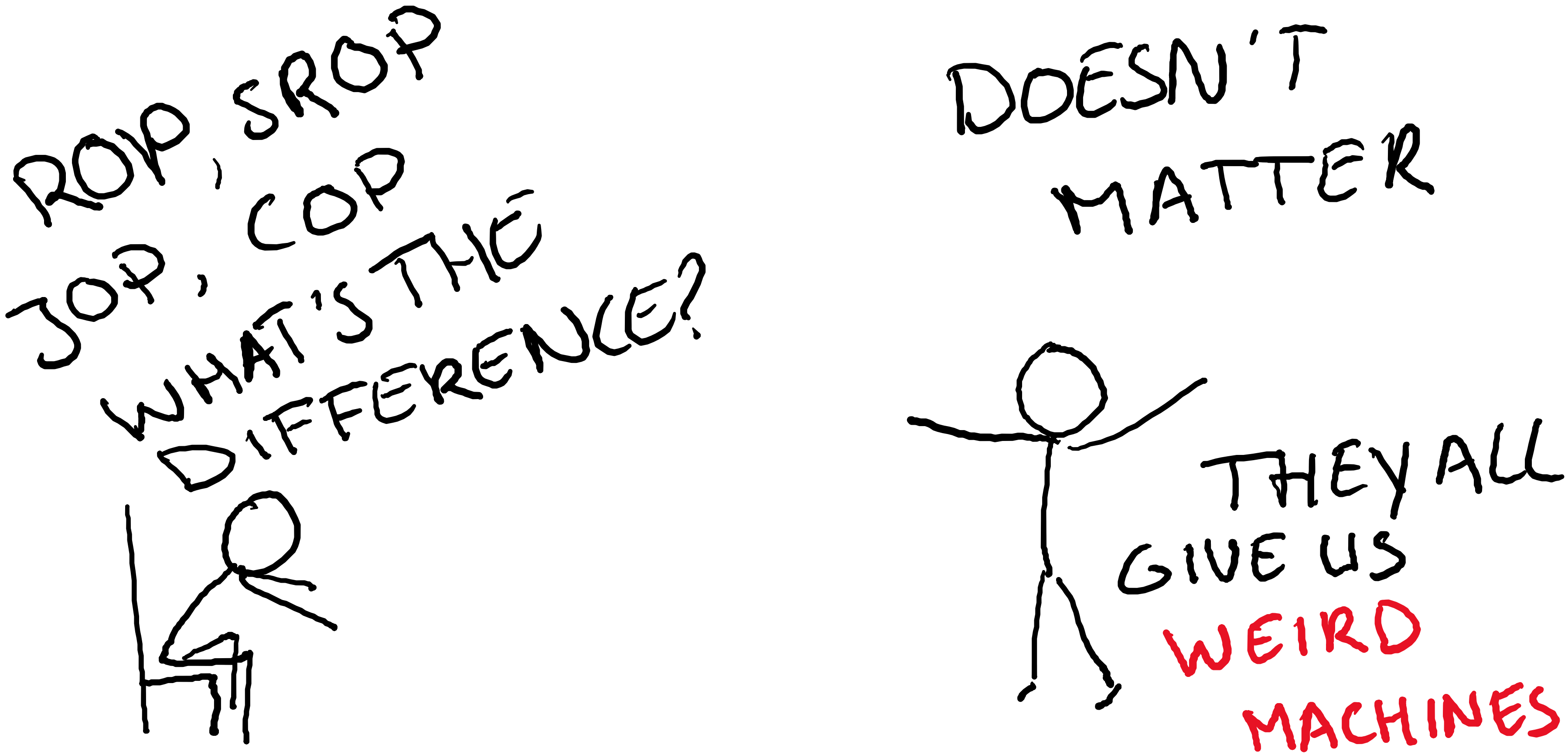
- Always present
- Sometimes at fixed location

Works on all UNIXes

solaris

iOS

"Framing Signals", Security & Privacy, 2014


ROP, SROP
JOP, COP
WHAT'S THE DIFFERENCE?
DOESN'T MATTER
THEY ALL GIVE US WEIRD MACHINES

Weird machines en Alan Turing

"What can be calculated?"

"Is it a browser of a flappy bird?"

We don't want

Turing Completeness

We want

Restrictive model

Existing defenses

Canaries

DEP

ASLR

Don't cut it!

Frantic Search For

Better Solutions

Like

CFI

CFI

top contender when other measures failed




Threat Model

Attackers divert control and jump to some **gadget** at arbitrary address (e.g., in the middle of function or even in an instruction)

Defense

Two *fundamental* requirements:

- locate code (gadgets)
- jump to gadgets

Control Flow Integrity

- Simple idea:
  - Allow only “legitimate branches and calls”

➔ That follow the control flow graph

Example CFG

```
void sort_1(int a[], int len)
{
    ...
    sort(a, len, less_than);
    ...
}
```

```
void sort_2(int a[], int len)
{
    ...
    sort(a, len, greater_than);
    ...
}
```

Example CFG

```
bool less_than(int x, int y);
```

```
bool greater_than(int x, int y);
```

```
void sort_1(int a[], int len)
{
    ...
    sort(a, len, less_than);
    ...
}
```

```
void sort_2(int a[], int len)
{
    ...
    sort(a, len, greater_than);
    ...
}
```

Example CFG

```
bool less_than(int x, int y);
```

```
bool greater_than(int x, int y);
```

```
bool sort(int a[], int len, comp_func_t fptr)
{
    ...
        if (fptr(a[i], a[i+i]))
            ...
    ...
}
```

```
void sort_1(int a[], int len)
{
    ...
    sort(a, len, less_than);
    ...
}
```

```
void sort_2(int a[], int len)
{
    ...
    sort(a, len, greater_than);
    ...
}
```

Example CFG

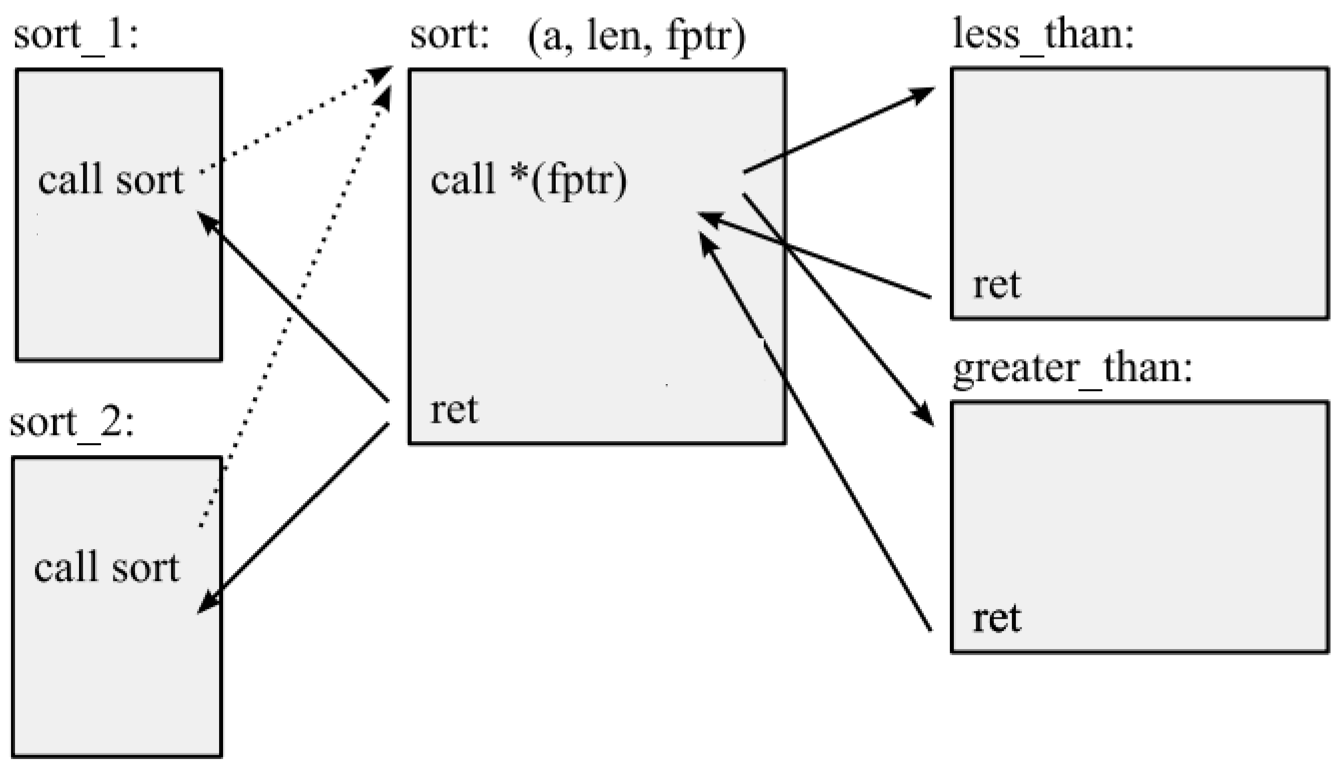
```
bool less_than(int x, int y);

bool greater_than(int x, int y);

bool sort(int a[], int len, comp_func_t fptr)
{
    ...
        if (fptr(a[i], a[i+i]))
            ...
    ...
}

void sort_1(int a[], int len)
{
    ...
    sort(a, len, less_than);
    ...
}

void sort_2(int a[], int len)
{
    ...
    sort(a, len, greater_than);
    ...
}
```

CFI example: we want to insert a check that *sort* can only return to just after the call in *sort_1* and just after the call to *sort_2*

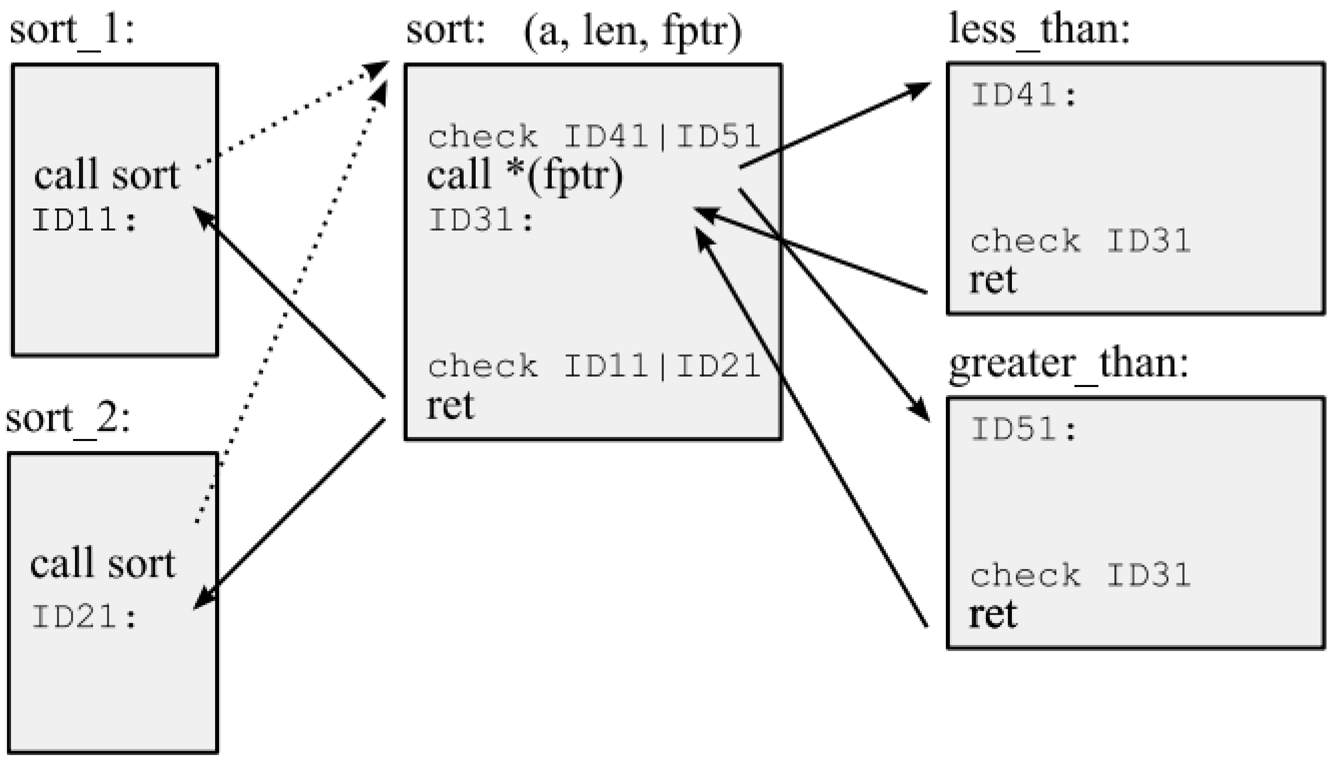
Similarly for other indirect branches (e.g., *fptr* can only land at *less_than* or *greater_than*

So we give the valid targets a "label" (or ID) and check that we do not branch anywhere else

Let us see that CFG again

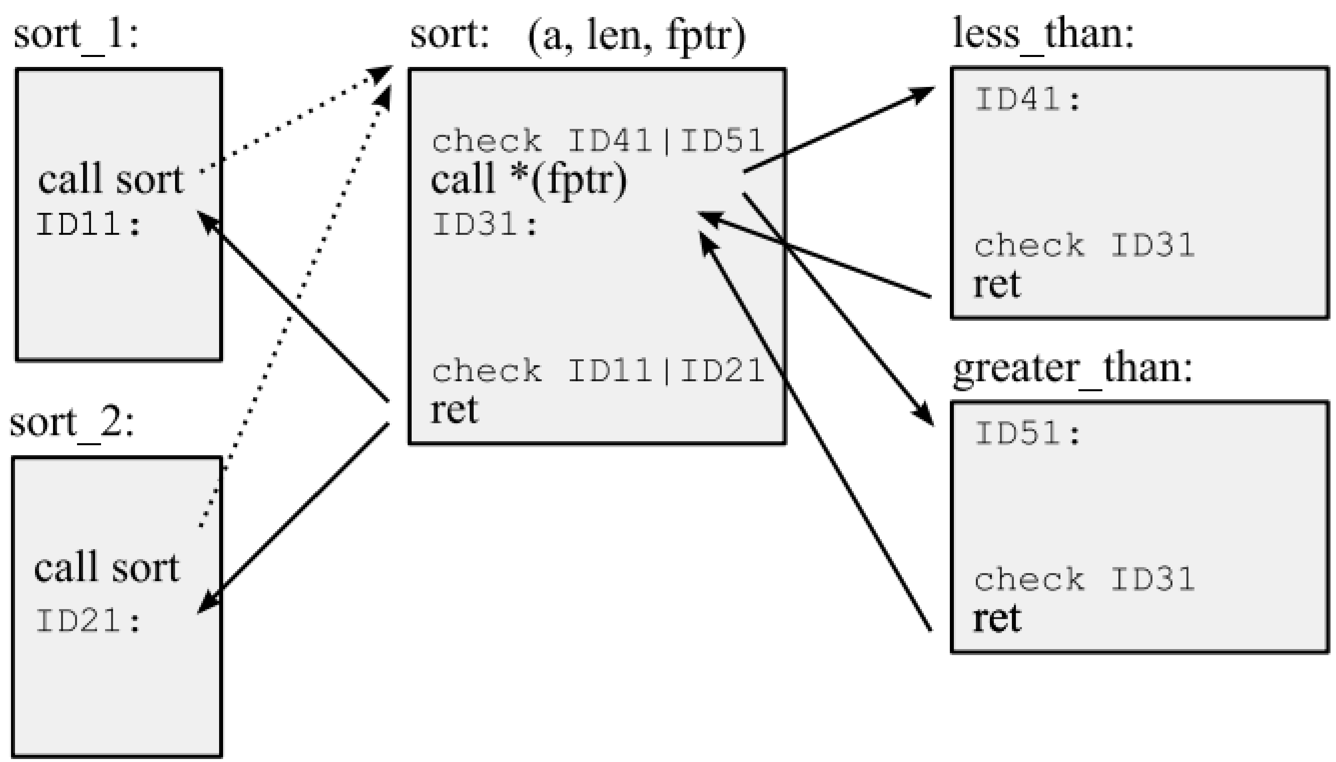


Let us see that CFG again



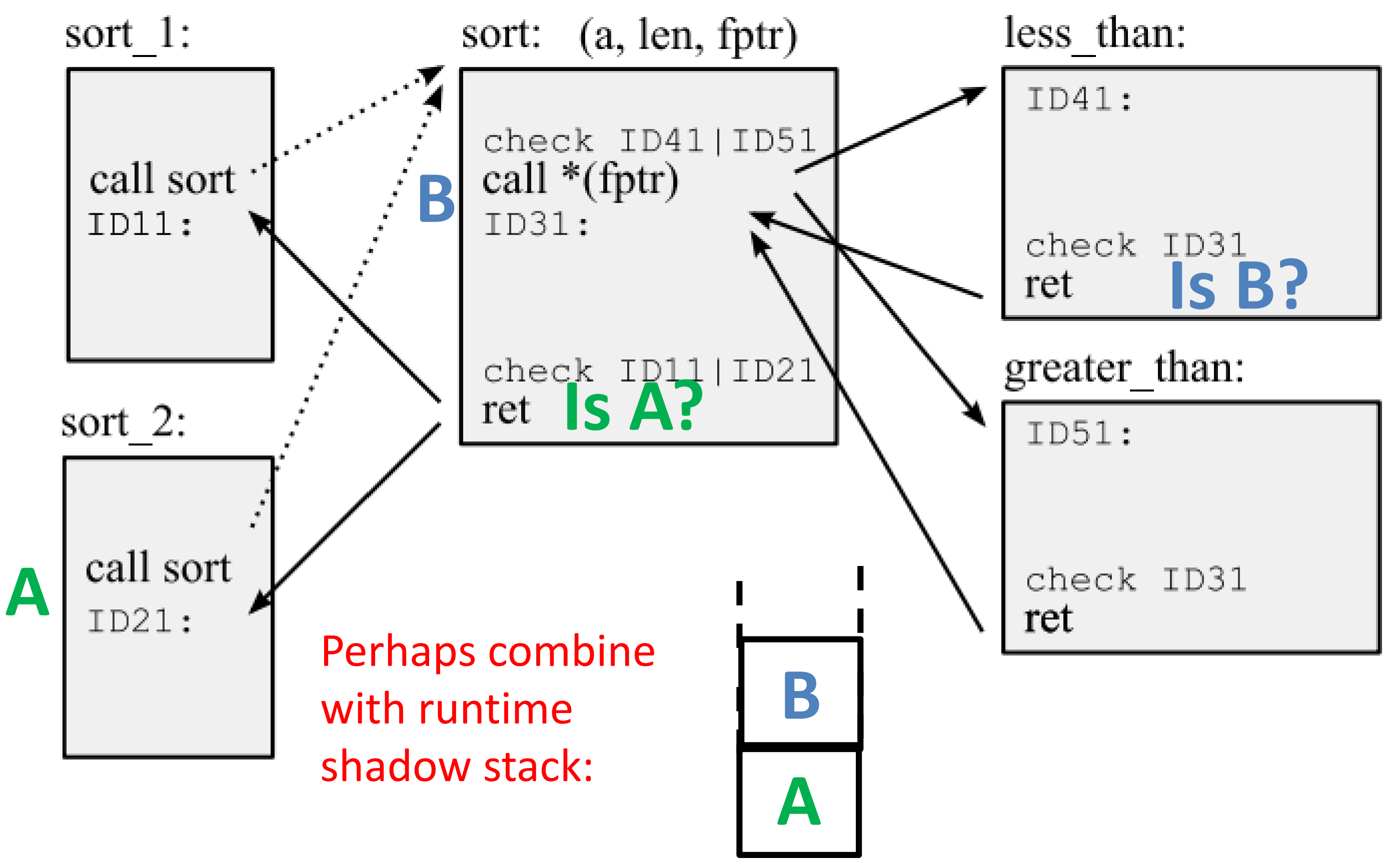
This is known as fine-grained CFI

Control-Flow Graph with ideal CFI



May be combined with a shadowstack

Control-Flow Graph with ideal CFI



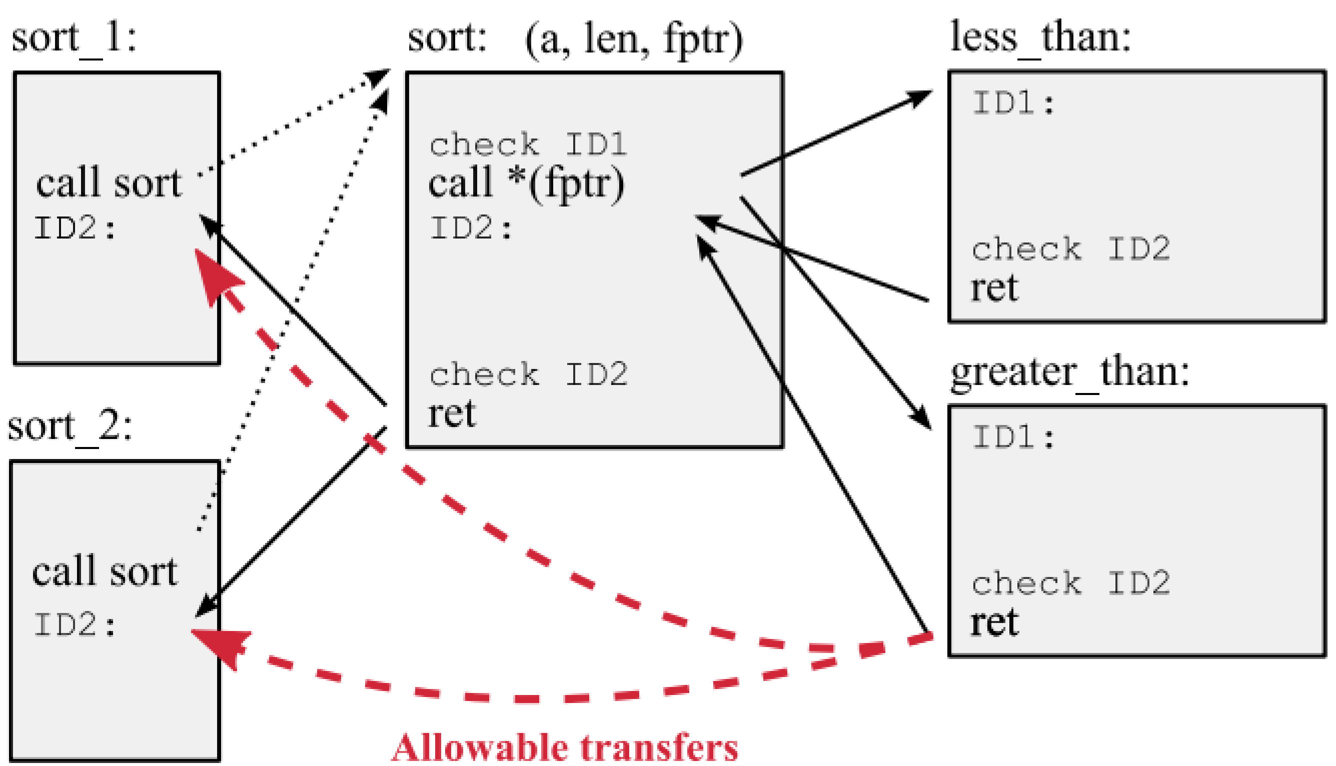
CFI in Practice

- Challenges for the adoption of the ideal CFI:
  - Requires precise Control-Flow Graph (CFG)
    - Requires source code or debug information
  - Has a non-negligible performance overhead
- So: loose CFI → practical (?)
  - Uses only a few labels
  - Applicable to binary-only software

Coarse-grained CFI supported by major compilers
Windows 10, Edge, … → all compiled with CFI!

Loose CFI
("coarse grained")

Control-Flow Graph with loose CFI



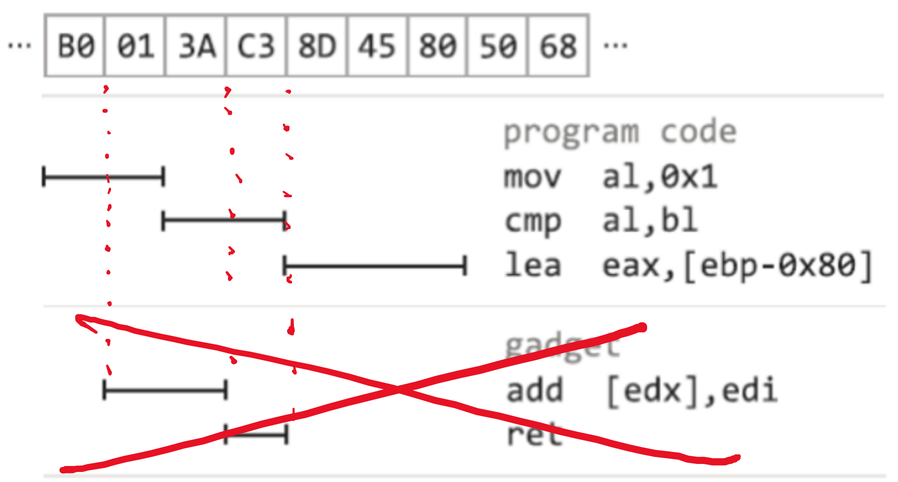
Still powerful

ROP

– can no longer use "misaligned gadgets"

– can only call legitimate function entry points

– and return to legitimate call sites



Eliminates

98%

of gadgets

Question

Is that enough?

Allowable transfers in practical CFI

Examples of coarse grained CFI implementations

Target:	**Abadi CFI (1 ID)**	**CCFIR (3 IDs)**	**bin-CFI (2 IDs)**
Return addresses	All indirect transfers	All *ret* instructions	*ret* & indirect *jmp* instructions
Return addresses in sensitive functions		*ret* instructions in sensitive functions	
Exported functions		Indirect *call* & *jmp* instructions	Indirect *call* instructions
Sensitive functions		X	

Allowable transfers in practical CFI

Examples of coarse grained CFI implementations

Target:	Abadi CFI (1 ID)	CCFIR (3 IDs)	bin-CFI (2 IDs)
Return addresses	All indirect transfers	All *ret* instructions	*ret* & indirect *jmp* instructions
Return addresses in sensitive functions		*ret* instructions in sensitive functions	
Exported functions		Indirect *call* & *jmp* instructions	Indirect *call* instructions
Sensitive functions		X	

Question: is this enough to stop ROP?

Phrased differently

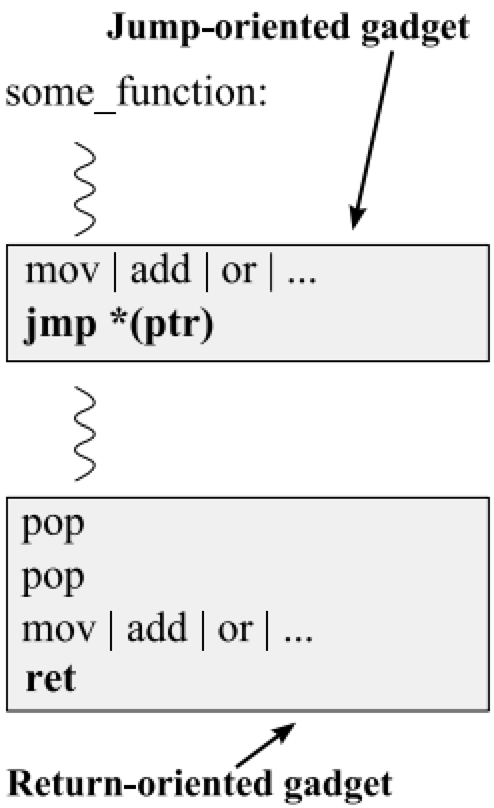
What gadgets are left?

Enes Goktas in 2005

Phrased differently

What gadgets are left?

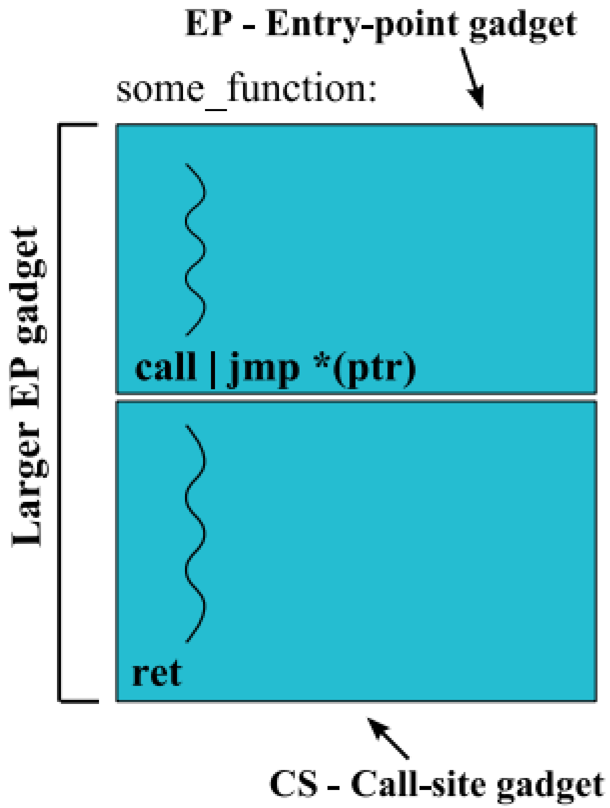
“Old” ROP gadgets



New gadgets!

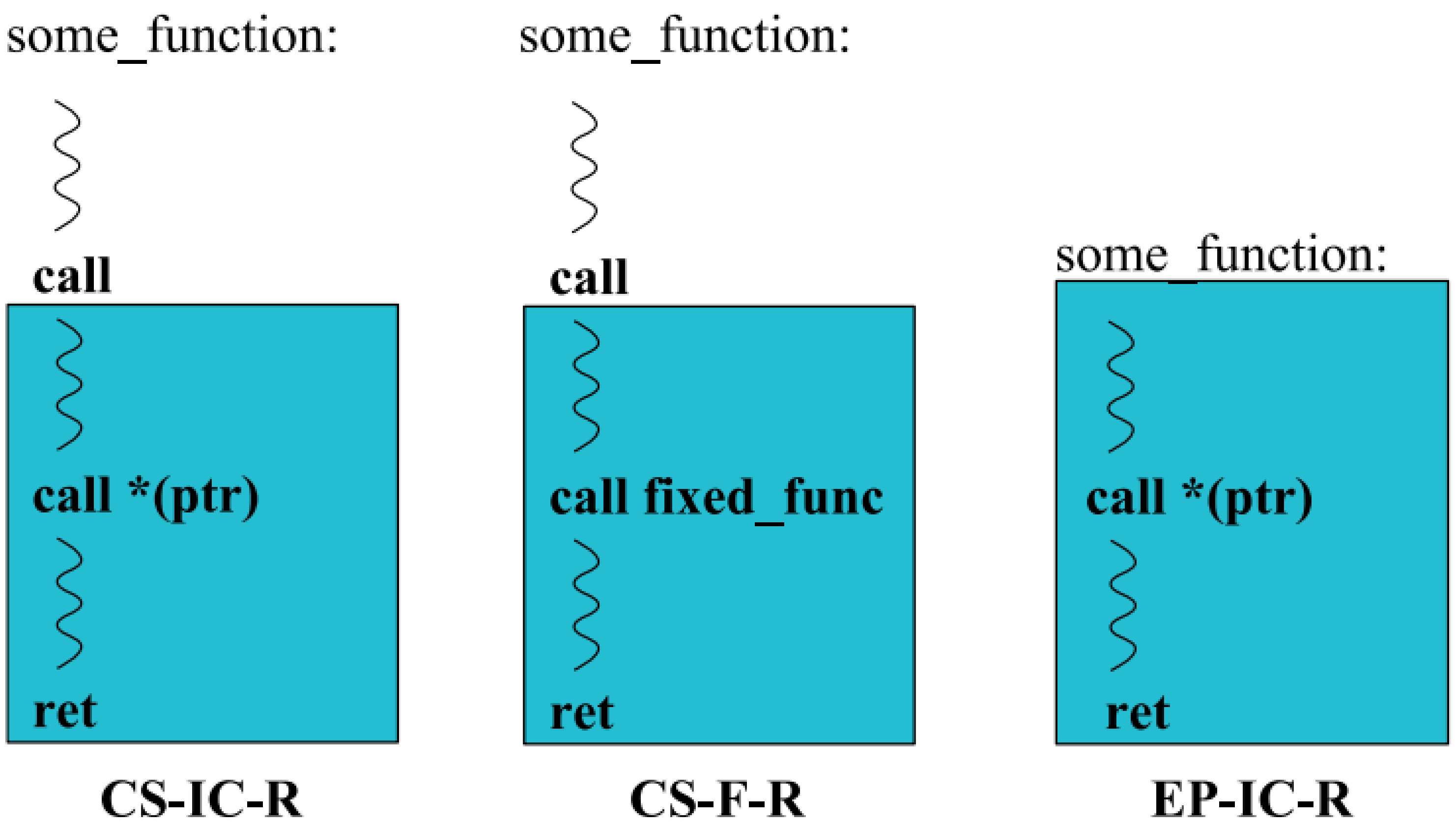
Entry point gadgets

Call site gadgets



Skip details

Calling functions



Proof of concept

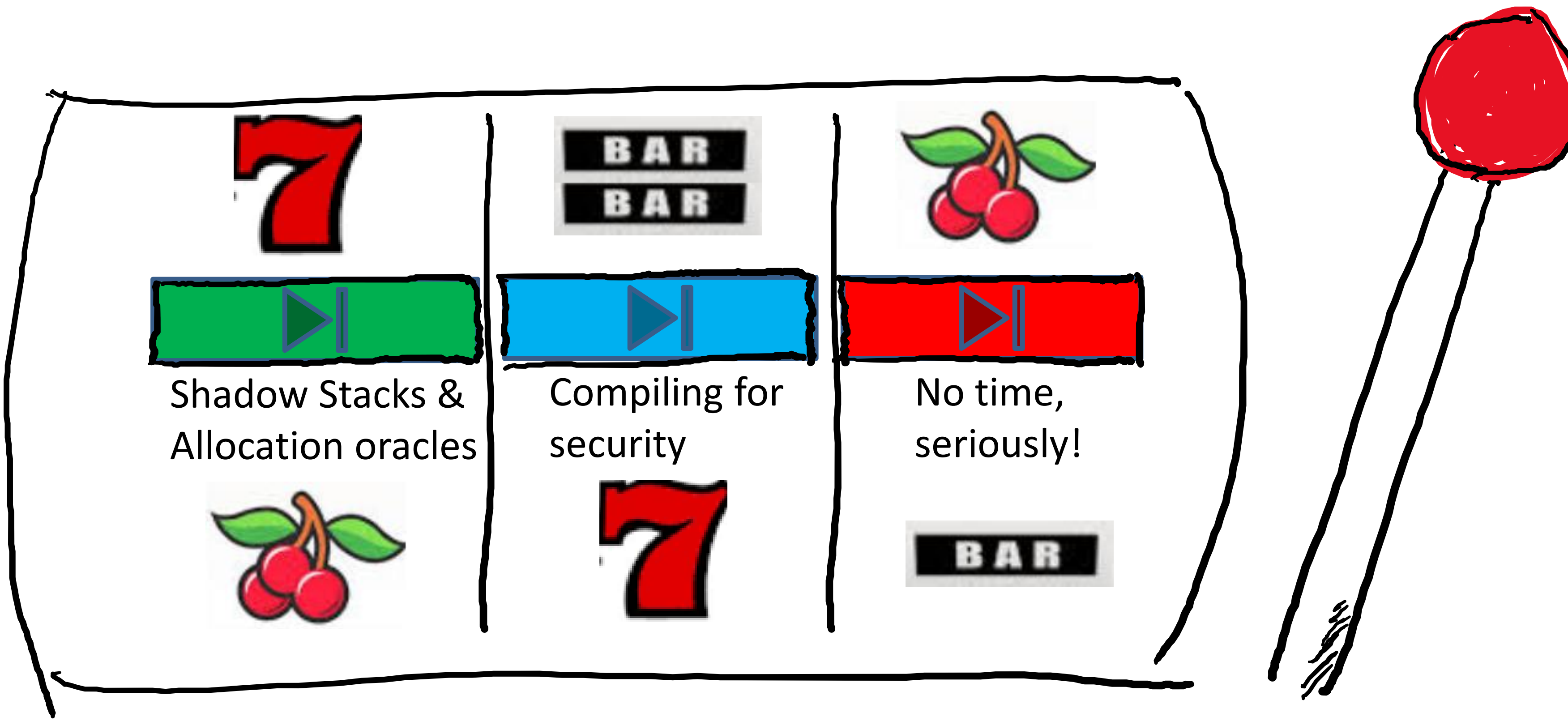
- Exploit in Internet Explorer 8
  - ASLR and DEP in place
  - CCFIR in place
- Vulnerability: heap buffer overflow (CVE-2012-1876)
  - Spray the heap with desired exploitation data
  - Trigger vulnerability
  - Get control of an indirect jmp
  - Execute linked gadgets

Threat Model

Attackers divert control and jump to some **gadget** ~~at arbitrary address (e.g., in the middle of function or even in an instruction)~~

Useless?

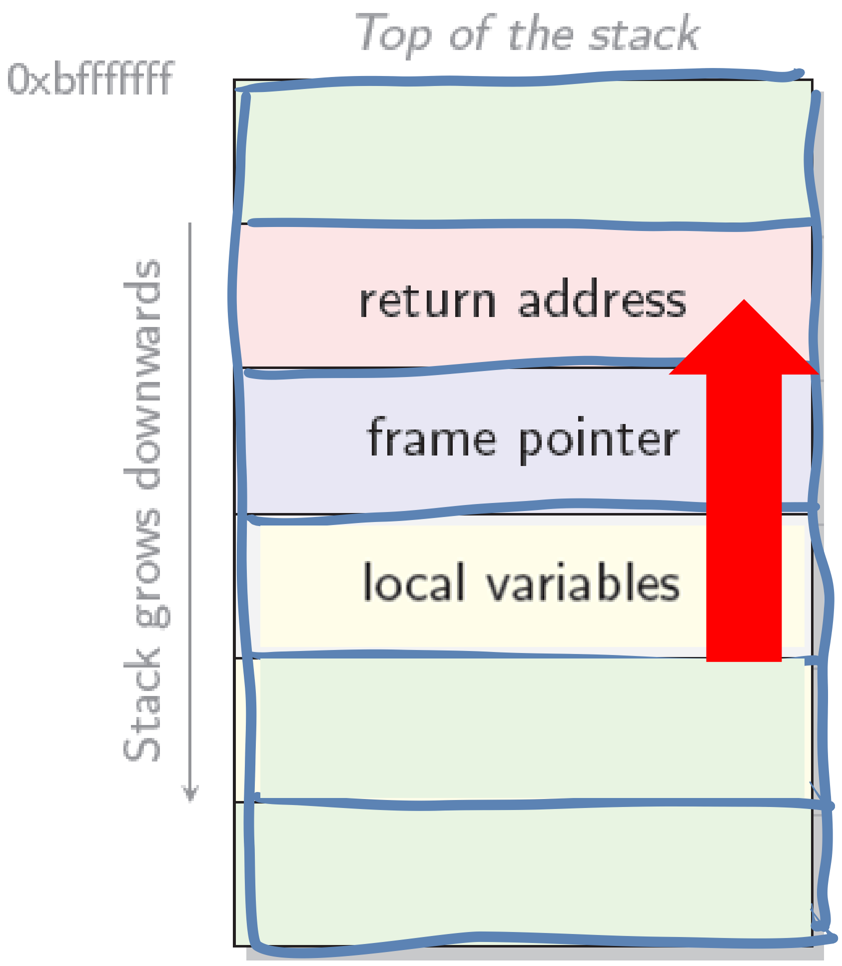
- No! It really raises the bar for attackers
- Even if there is still some wiggle room left


BAR
BAR
Shadow Stacks & Allocation oracles
Compiling for security
No time, seriously!
BAR

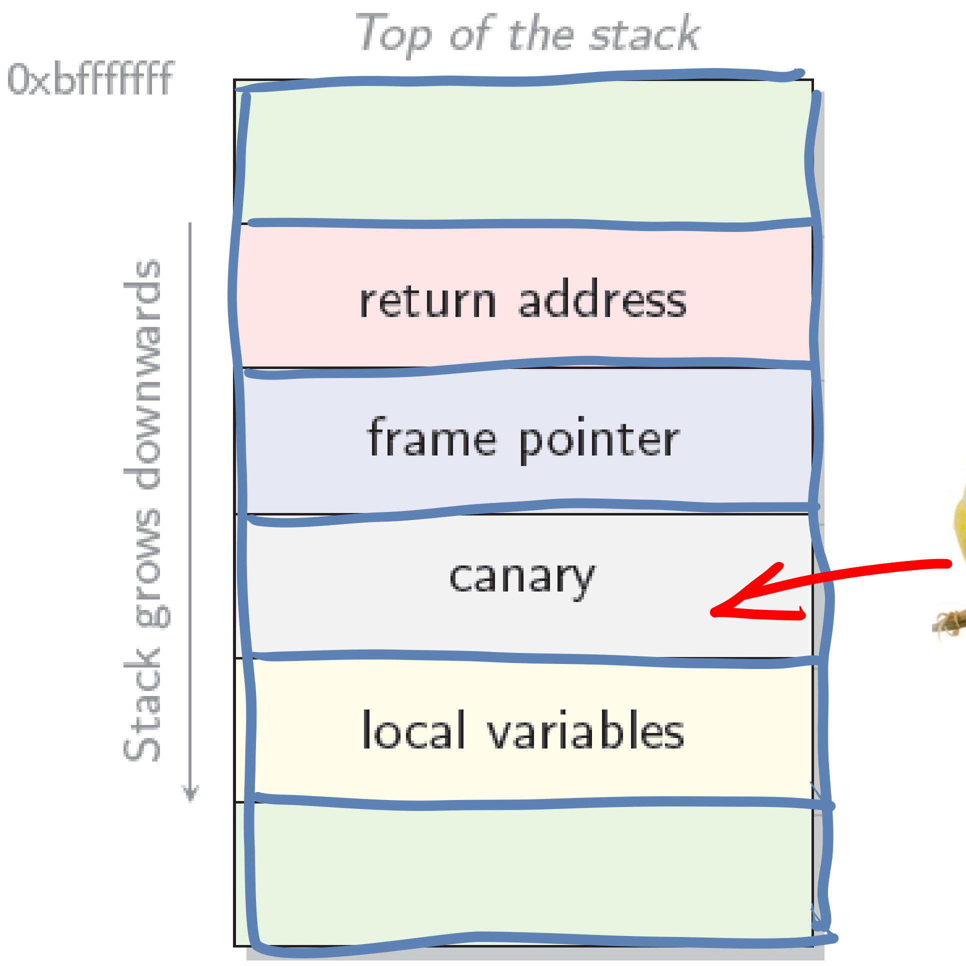
Incidentally, shadow stacks

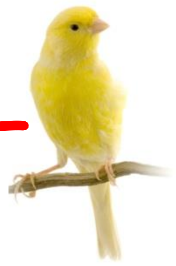
- Are like CFI
- But very precise
  - Take context into account

Remember: stacks were a problem

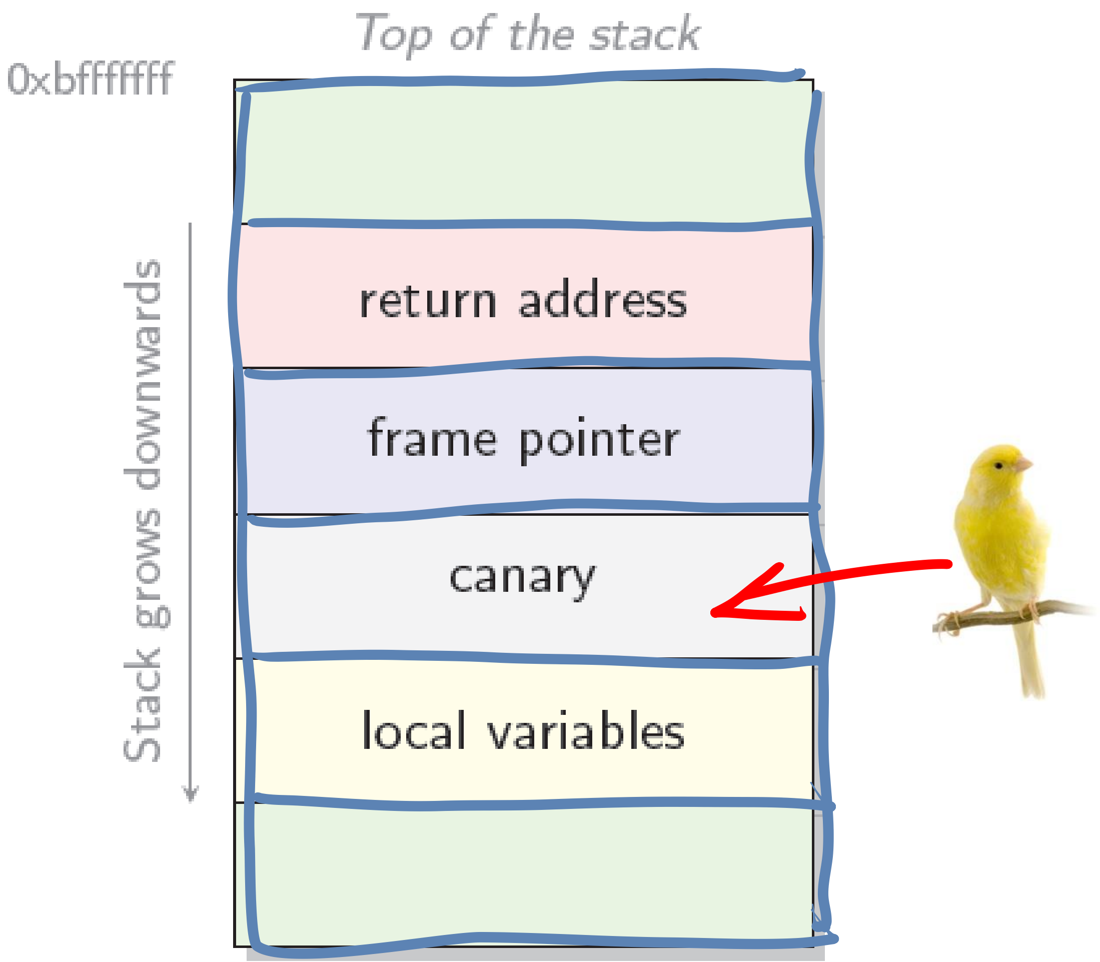


Then we added canaries



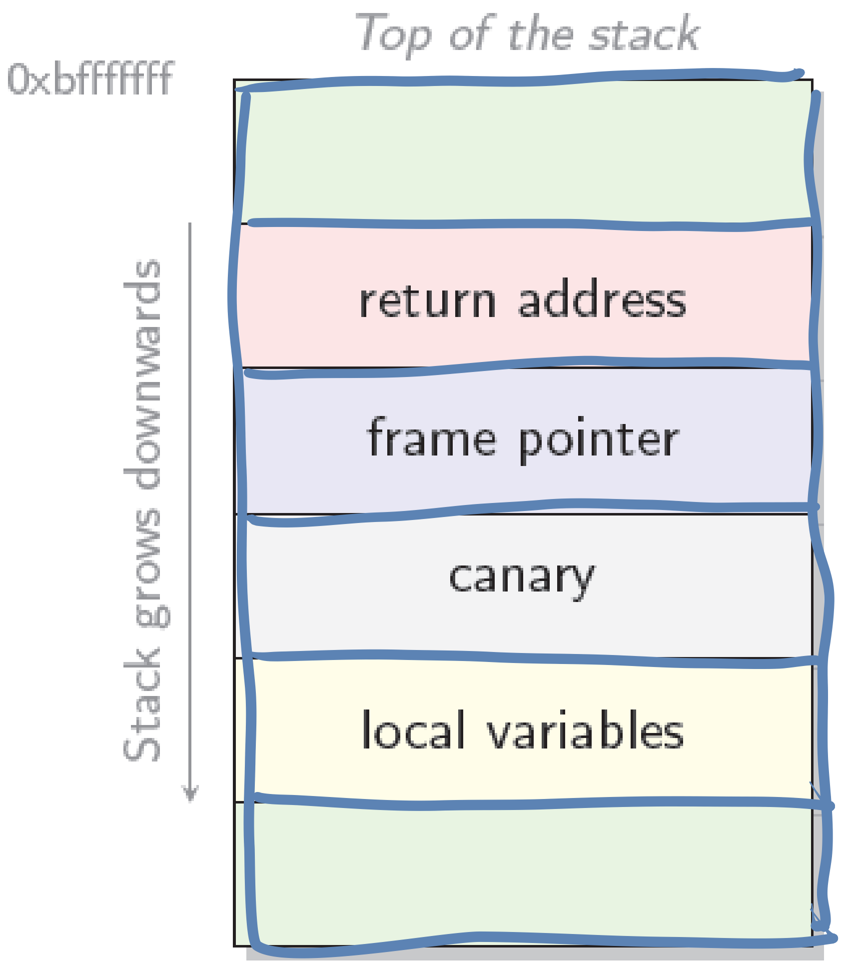


Canaries not enough



VU University Amsterdam

So: stacks still a problem



what if

we moved the

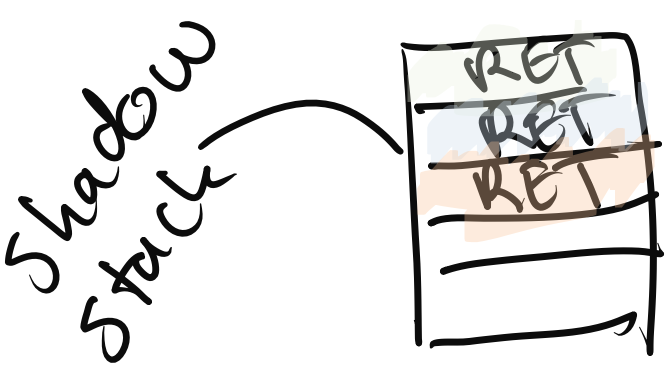
sensitive data out?

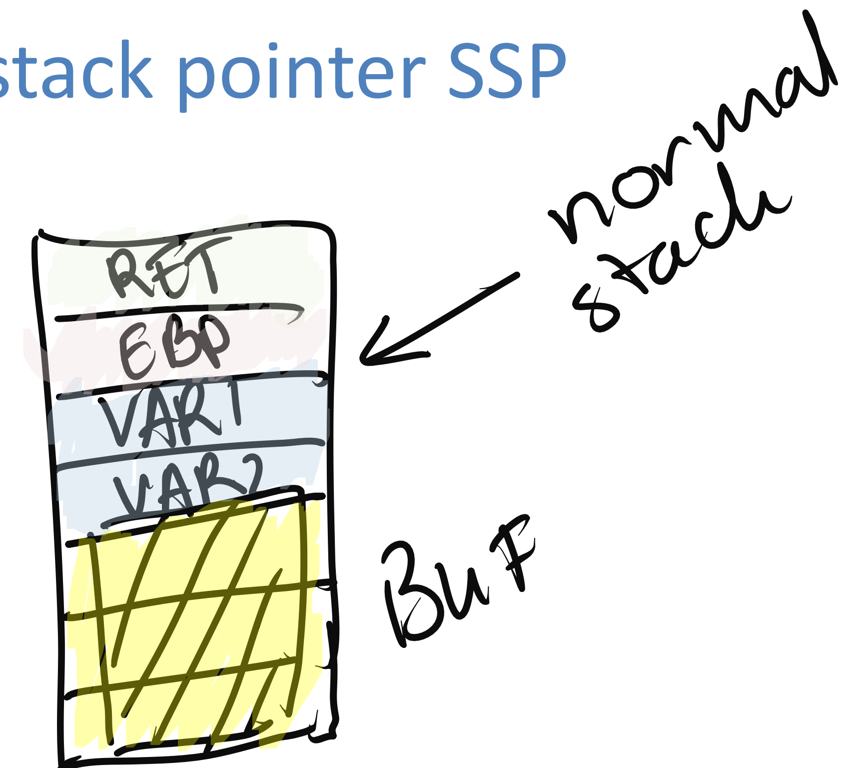
Shadow Stack

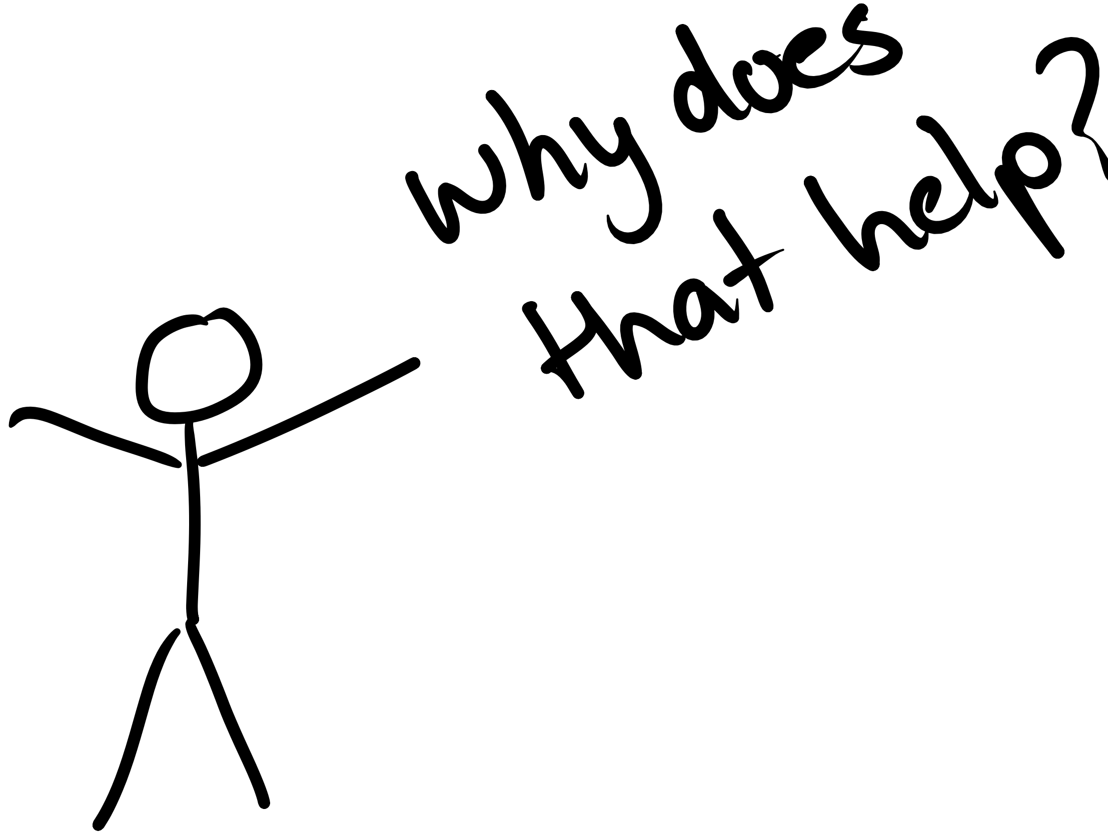
- Move sensitive data (e.g., return addresses) on separate stack
  - Known as shadow stack
- Keep separate shadow stack pointer SSP

Shadow Stack

- Move sensitive data (e.g., return addresses) on separate stack
  - Known as shadow stack
- Keep separate shadow stack pointer SSP





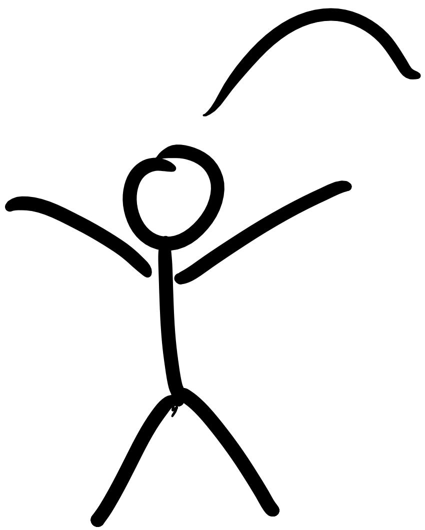

why does that help?

Let's see the way it works

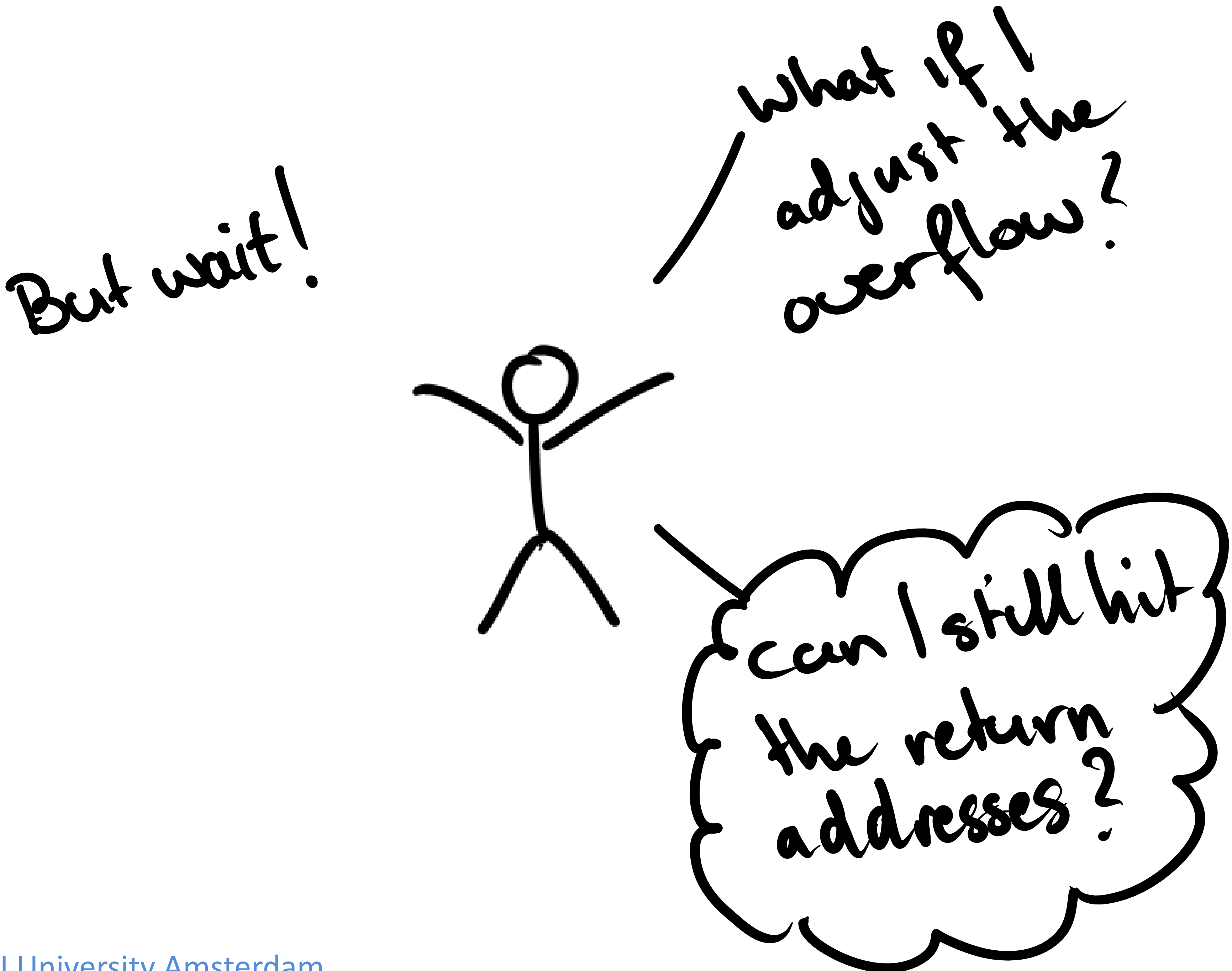
- On function call:
  - Save return address on shadow stack
- On return:
  - Use return address on shadow stack

Threat Model

Attackers smash the stack (using contiguous or non-contiguous corruption) and modify return addr



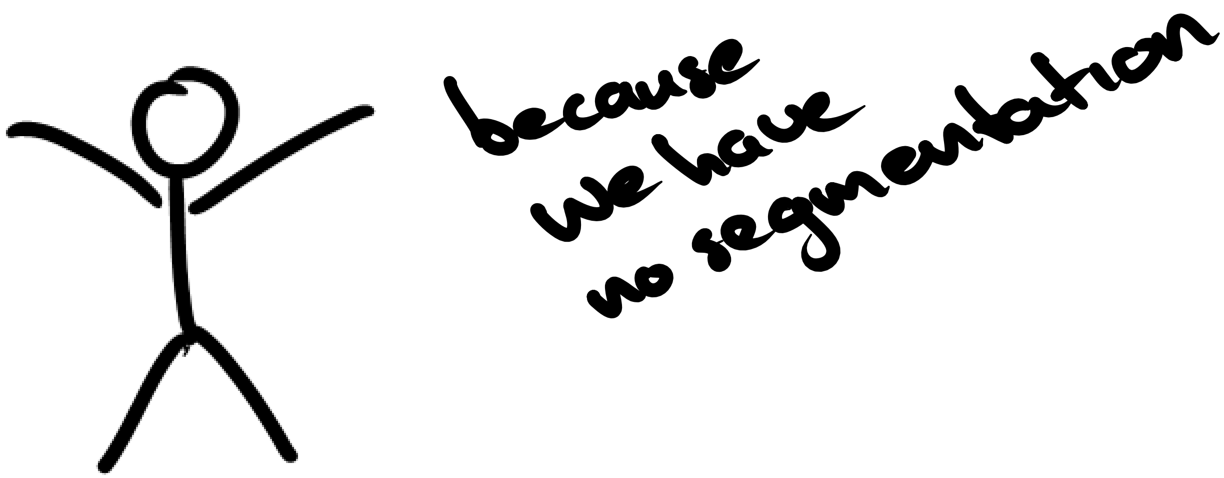
Ah! so an overflow on the regular stack will not corrupt the return addresses


But wait!
What if I adjust the overflow?
can I still hit the return addresses?

We have to make sure

Shadow Stack is protected

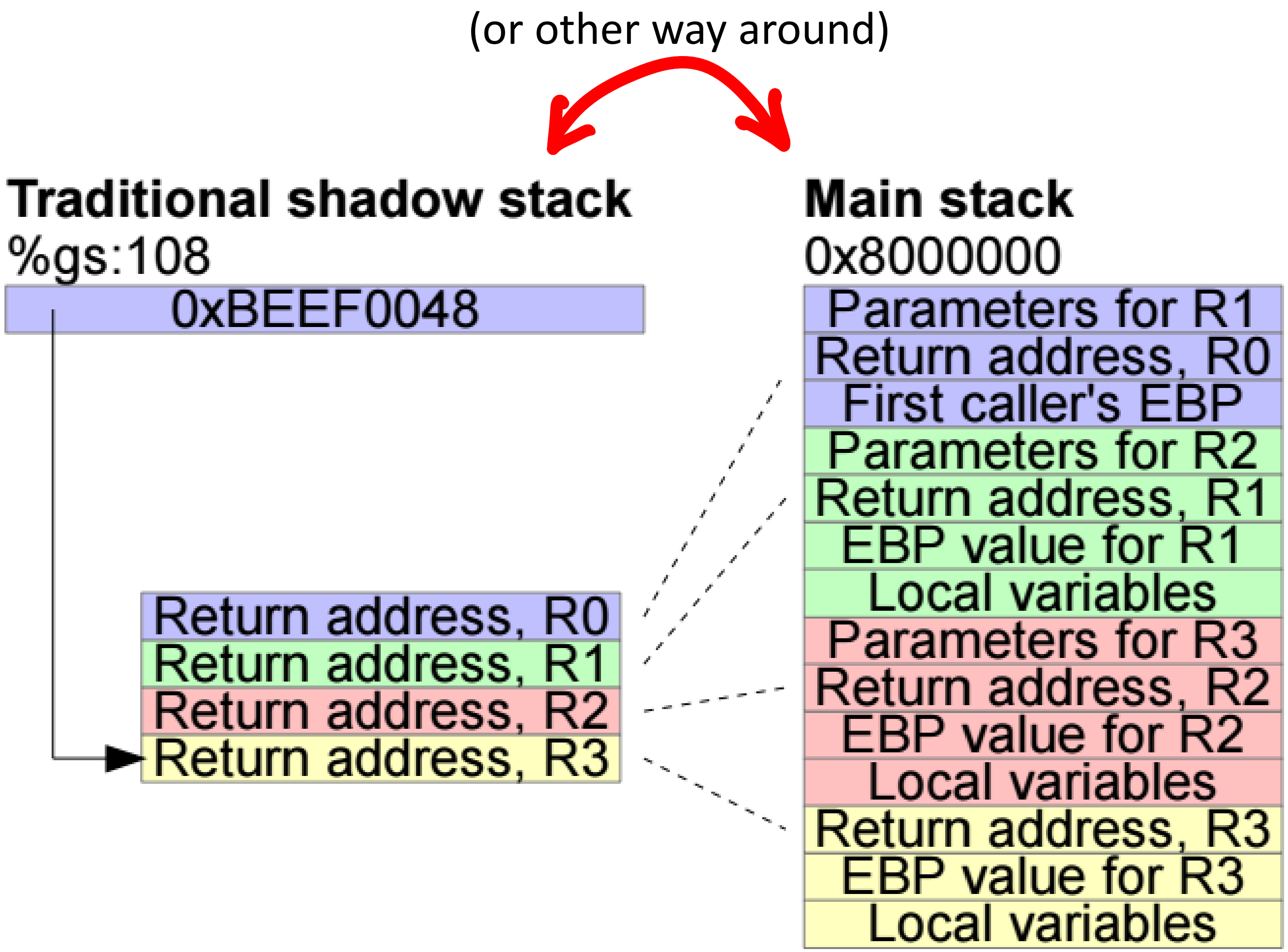
On 64bit machines: often done by means of information hiding



Real hiding

- No pointer in memory should refer to the secret location of the shadow stack
  - Only dedicated register points to the shadow stack

Stacks were a problem



New prologue

```
SUB $4, %gs:108    # Decrement SSP
MOV %gs:108, %eax  # Copy SSP into EAX
MOV (%esp), %ecx   # Copy ret. address into
MOV %ecx, (%eax)   #     shadow stack via ECX
```

Figure 2: Prologue for traditional shadow stack.

New epilogue (2 options)

```
MOV %gs:108, %ecx # Copy SSP into ECX
ADD $4, %gs:108   # Increment SSP
MOV (%ecx), %edx  # Copy ret. address from
MOV %edx, (%esp)  #     shadow stack via EDX
RET
```

Figure 3: Epilogue for traditional shadow stack (overwriting).

```
MOV %gs:108, %ecx
ADD $4, %gs:108
MOV (%ecx), %edx
CMP %edx, (%esp) # Instead of overwriting,
JNZ abort        #     we compare
RET
abort:
    HLT
```

Figure 4: Epilogue for traditional shadow stack (checking).

In a way…

- Return address serves as canary!

Threat Model

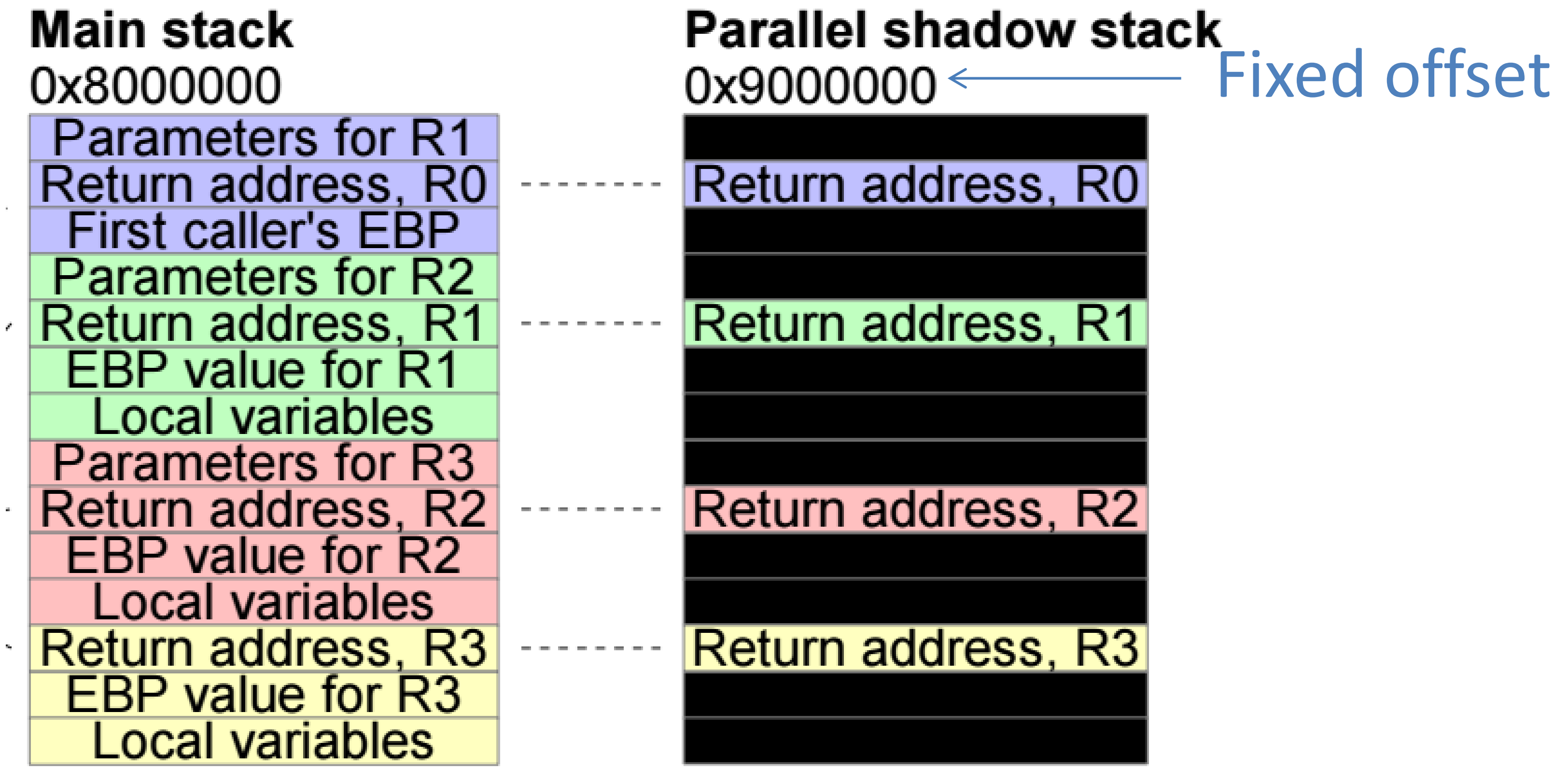
Attackers smash the stack (using contiguous or non-contiguous corruption) and modify return addr

Attackers can't touch shadow stacks

Anyway, implementation is slow

- Can we make it faster?

Keep the stacks “the same”



Prologue / Epilogue

```
POP 999996(%esp) # Copy ret addr to shadow stack
SUB $4, %esp # Fix up stack pointer (undo POP)
```

Figure 7: Prologue for parallel shadow stack.

```
ADD $4, %esp # Fix up stack pointer
PUSH 999996(%esp) # Copy from shadow stack
```

Figure 8: Epilogue for parallel shadow stack.

Advantages

- No clobbering of registers
  - Can preserve even flags (by using LEA instead of ADD/SUB)
- Very few instructions needed

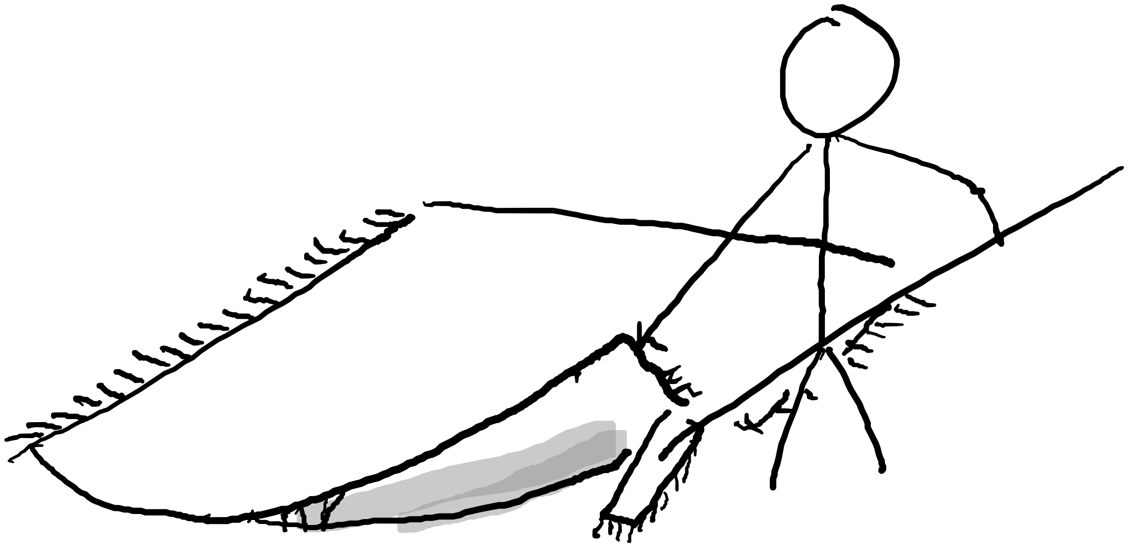
Overhead

- Traditional shadow stack : approx. 10%
- Minimalistic parallel shadow stack: approx. 3.5%

“a shadow stack, even when pared to its bare minimum (the overwriting, non-zeroing version of the parallel shadow stack), has non-negligible performance overhead, due to increased memory pressure”

[“The Performance Cost of Shadow Stacks and Stack Canaries”, AsiaCCS 2015]

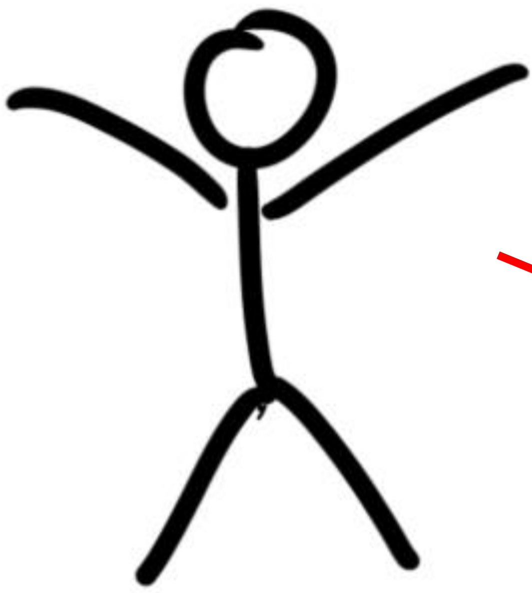
One thing we swept under the carpet



How do we protect the Shadow Stack?

- On x86 (32b): easy.
  - We have hardware segmentation.
- On x86-64: ah, well.
  - No hardware segmentation.
  - But: we have a whopping big address space.
  - How about squirrelling it away at some random location? Good luck finding that, attackers!

Information Hiding!

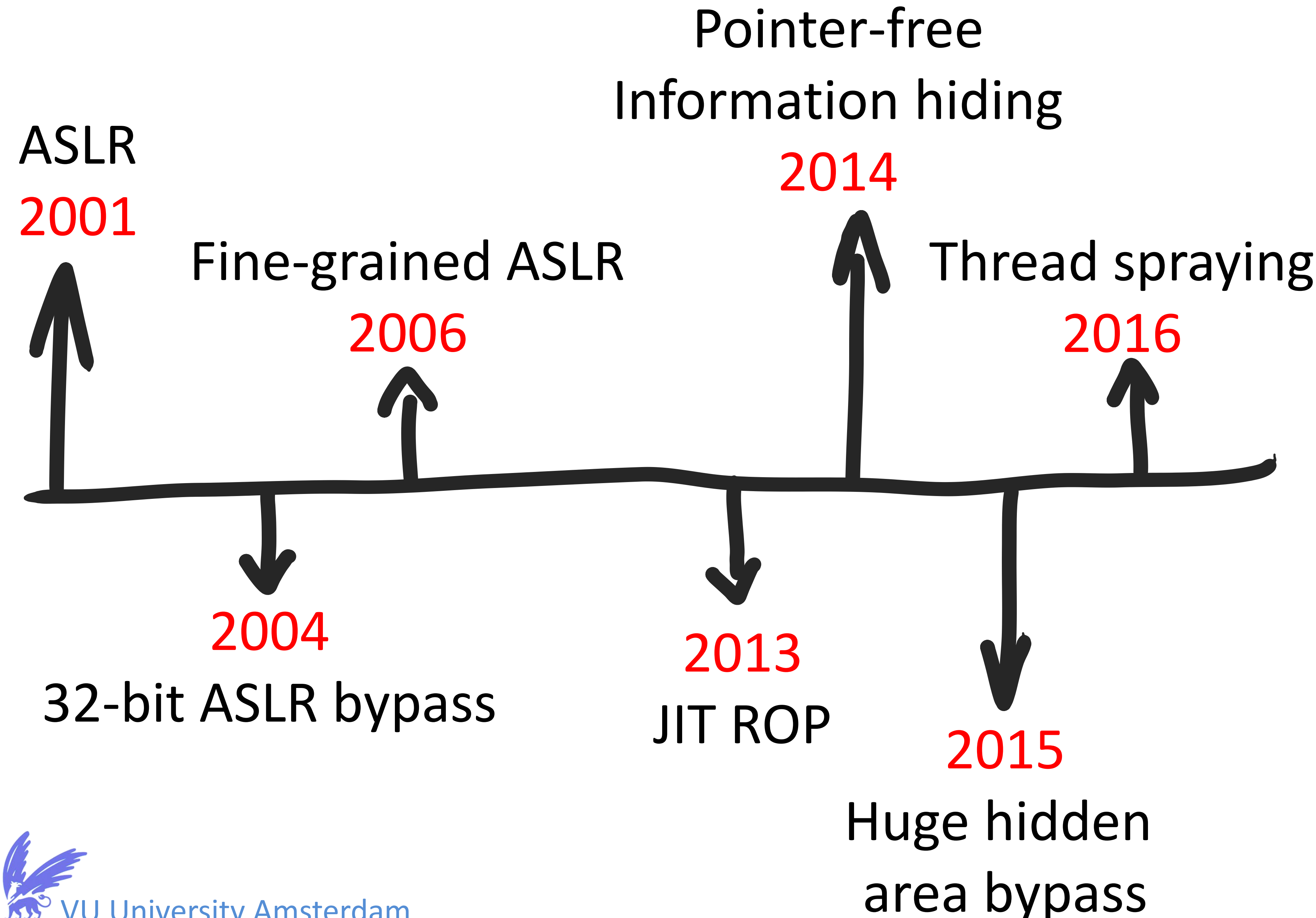


Relies on ASLR

ASLR today

- No longer a strong defense by itself
- But *pivotal* role in powerful new defenses:
  - Shadow stacks
  - Secure heap allocators
  - CPI (OSDI ’14)
  - StackArmor (NDSS ’15)
  - ASLR-guard (CCS ’15)
  - SafeStack (clang/llvm)
  - …

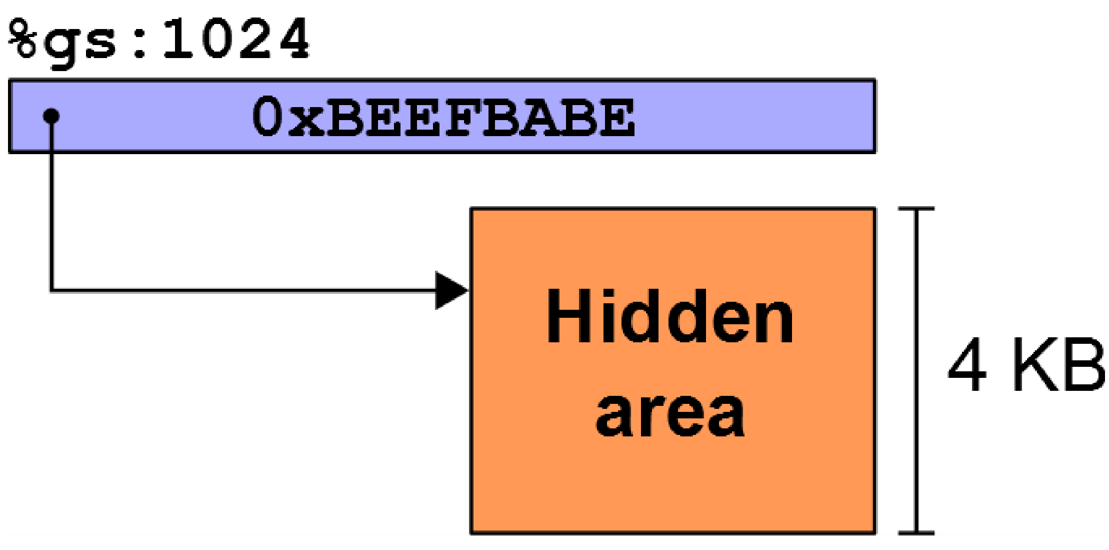
Evolution again



Ideal information hiding

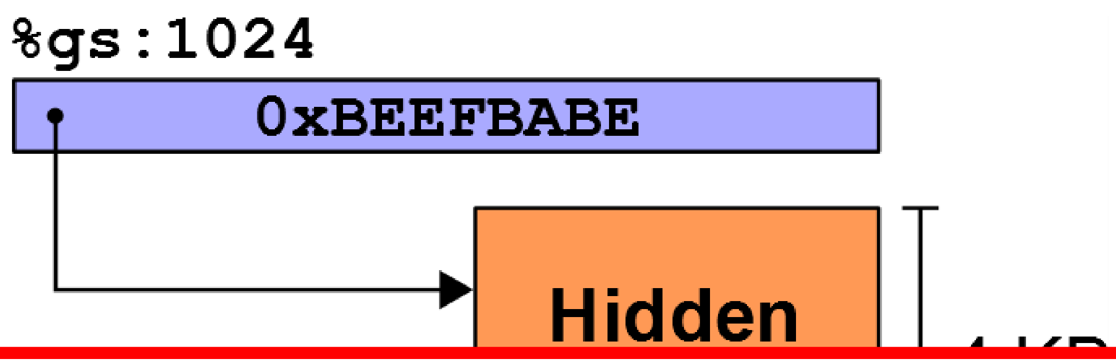
Leakage not possible

- The hidden area:
  - Has **no pointers** in memory referring to it
  - Is as **small** as possible
  - **Does not grow** during the execution



Ideal information hiding

- The hidden area:
  - Has **no pointers** in memory referring to it
  - Is as **small** as possible
  - **Does not grow** during the execution



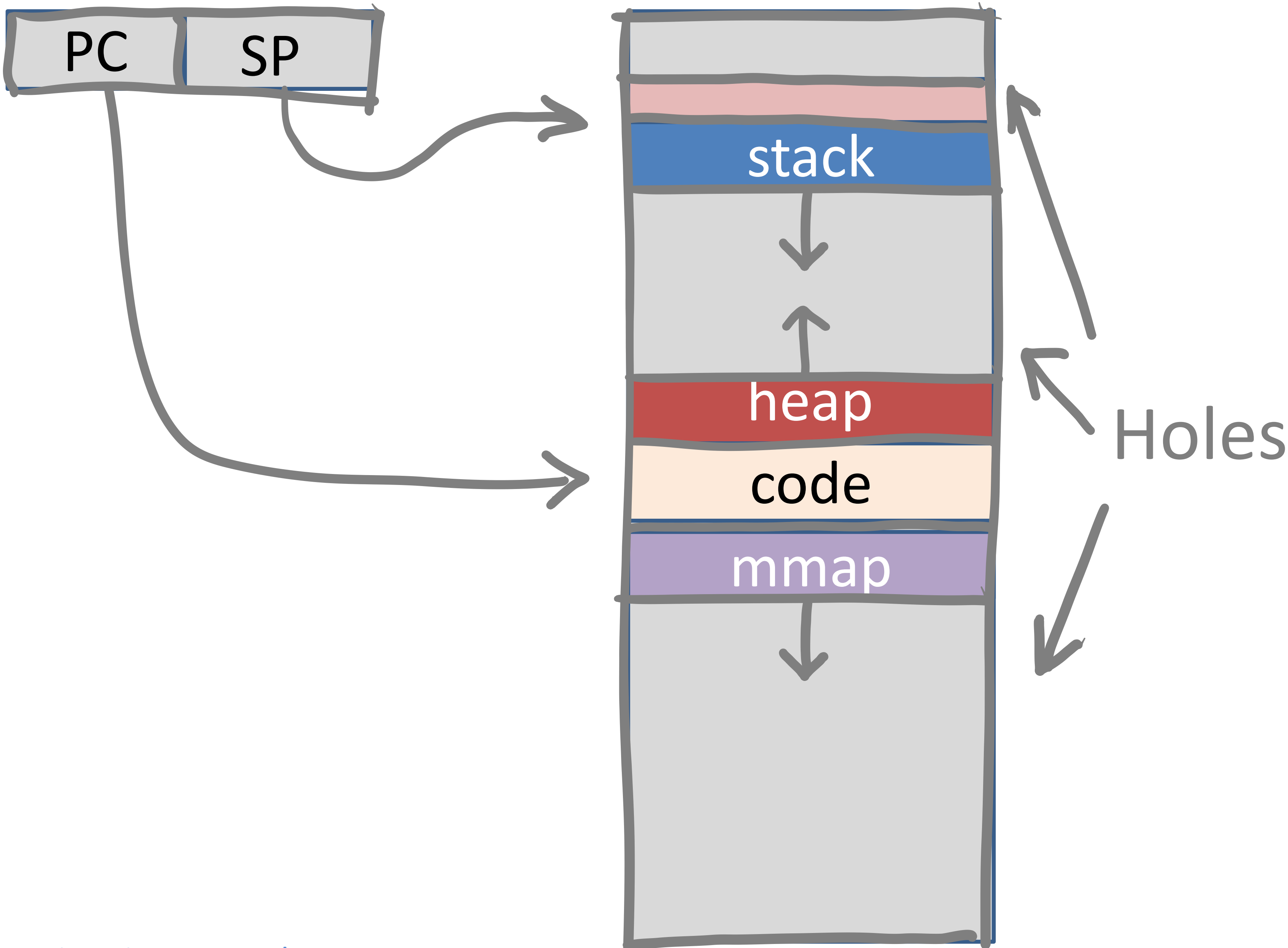
Threat model: arbitrary RW is okay!

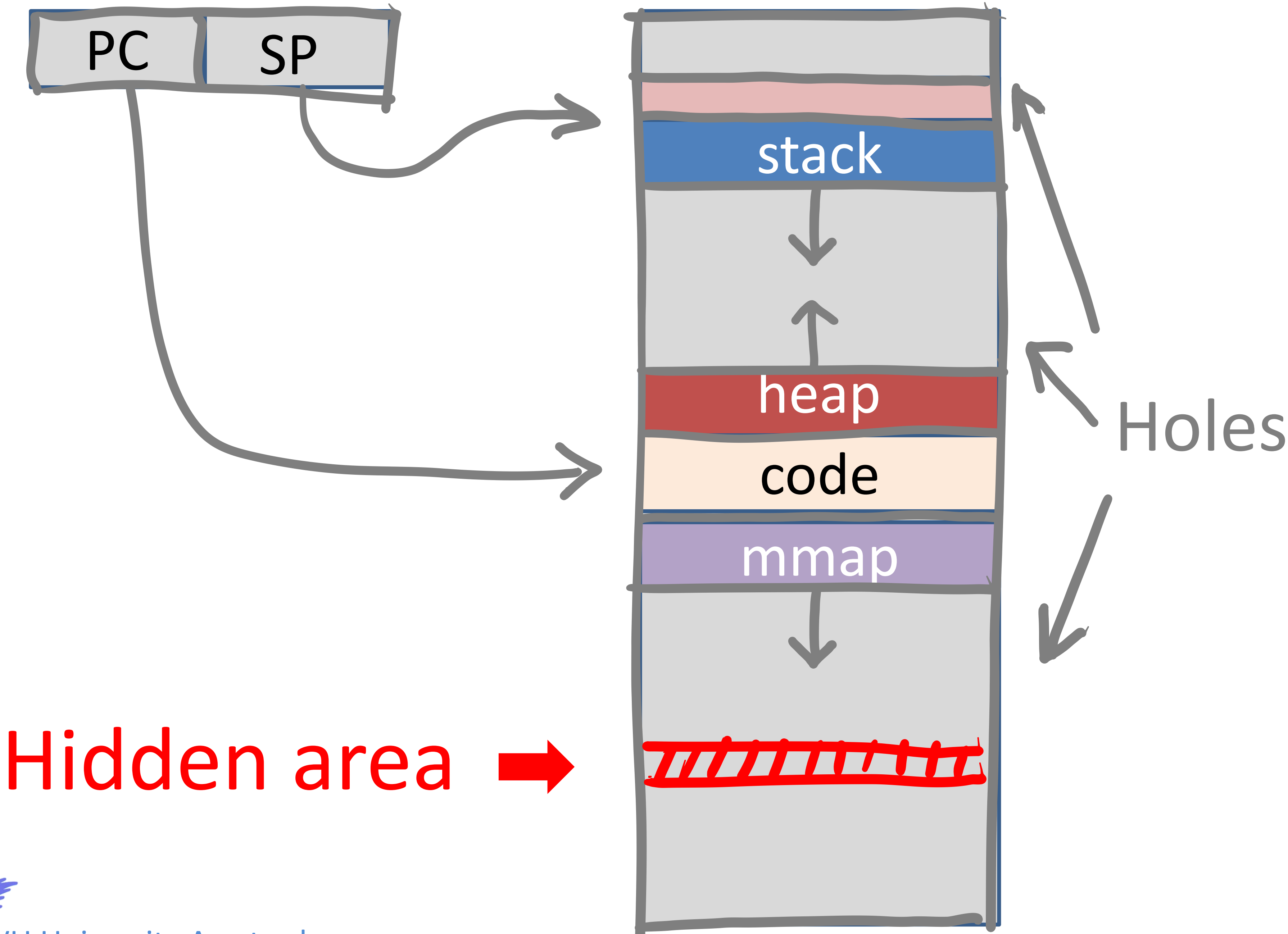
How SAFE?

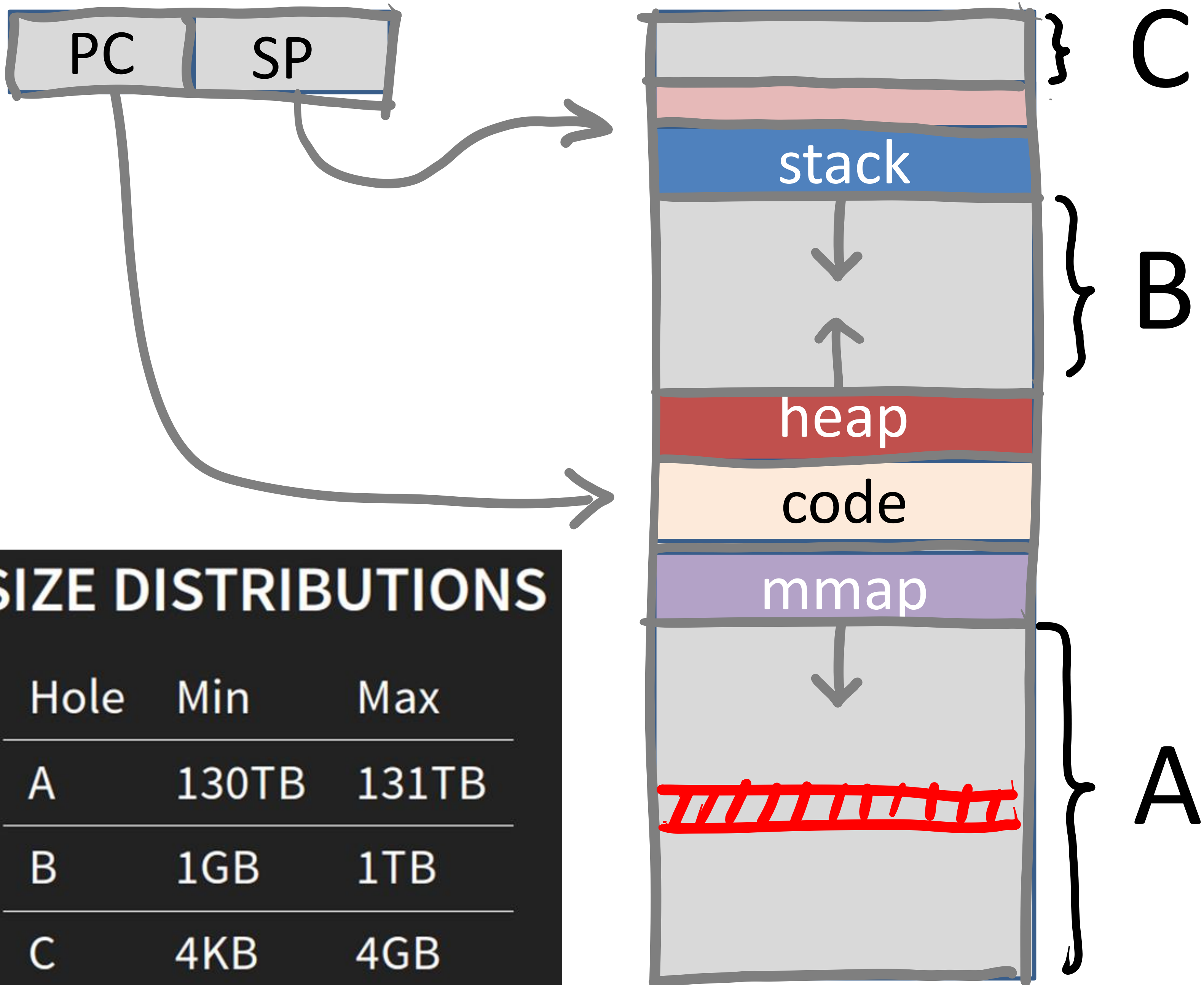
Let's have a look

at a Linux (PIE)

Address Space


PC
SP
stack
heap
code
mmap
Holes


PC
SP
stack
heap
code
mmap
Holes
Hidden area



SIZE DISTRIBUTIONS

Hole	Min	Max
A	130TB	131TB
B	1GB	1TB
C	4KB	4GB

Threat Model

Attackers cannot touch the shadow stacks (or any other info that is hidden in this address space)

Let’s not look for the hidden area

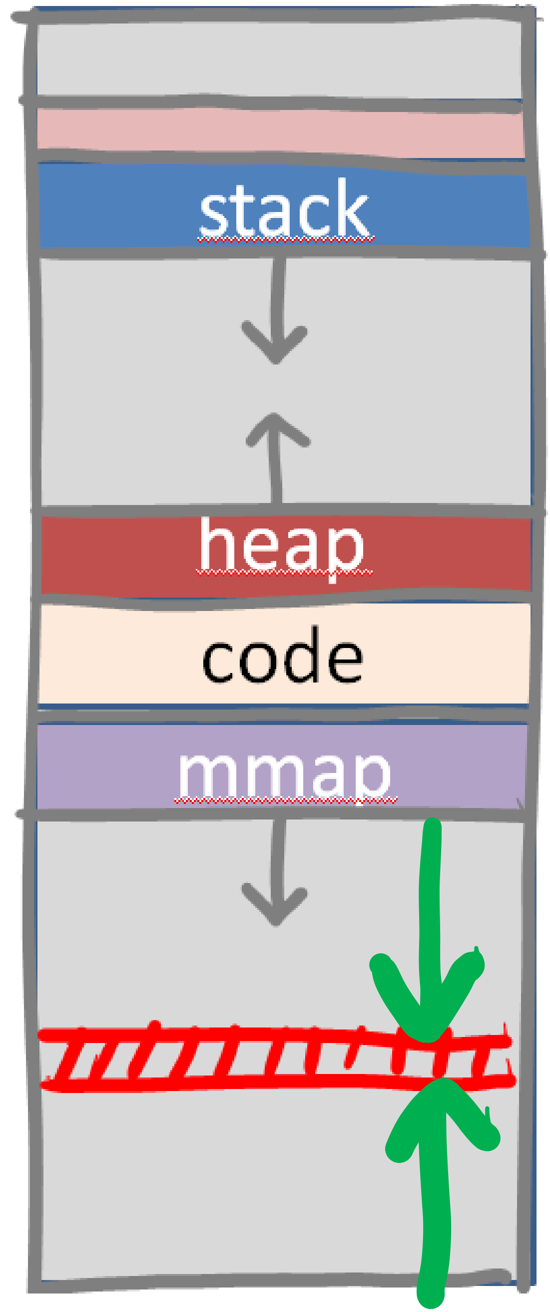
but for the holes!

Even if we remove all pointers

There is one "pointer" left: the **size** of the hole itself.

Then:

- Leak size of the largest hole
  → Infer hidden area location
- Not stored in user memory
  → Can't leak directly
- However, we can side-channel the kernel to spill the beans!

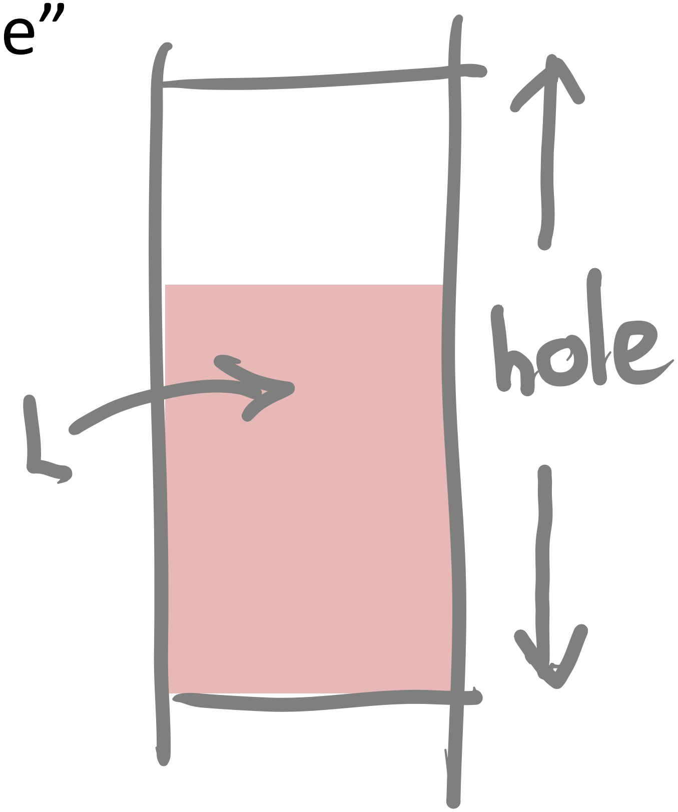


So look for the holes

- Intuition:
  - repeatedly allocate large chunks of memory of size **L** until we find the "right size"

Succeeds!

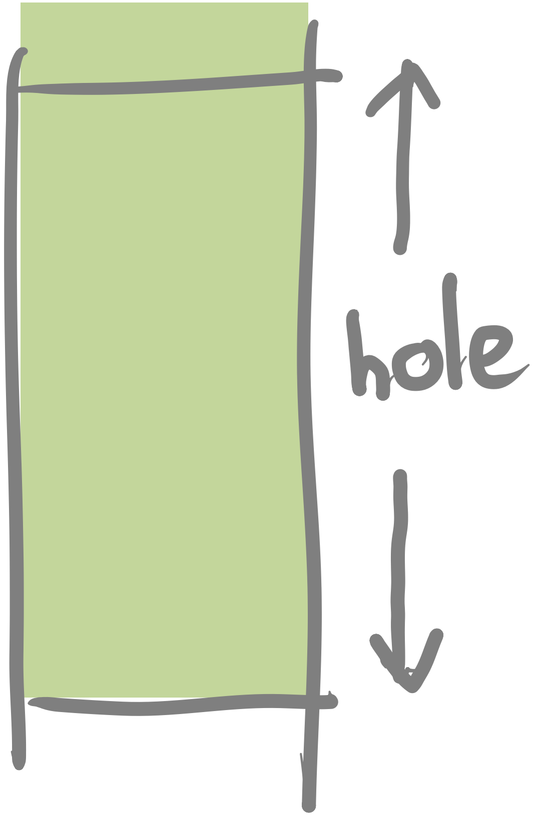
Sizeof(Hole) ≥ L



So look for the holes

- Intuition:
  – repeatedly allocate large chunks of memory of size **L** until we find the "right size"

Too large, alloc fails!
Sizeof(Hole) < L

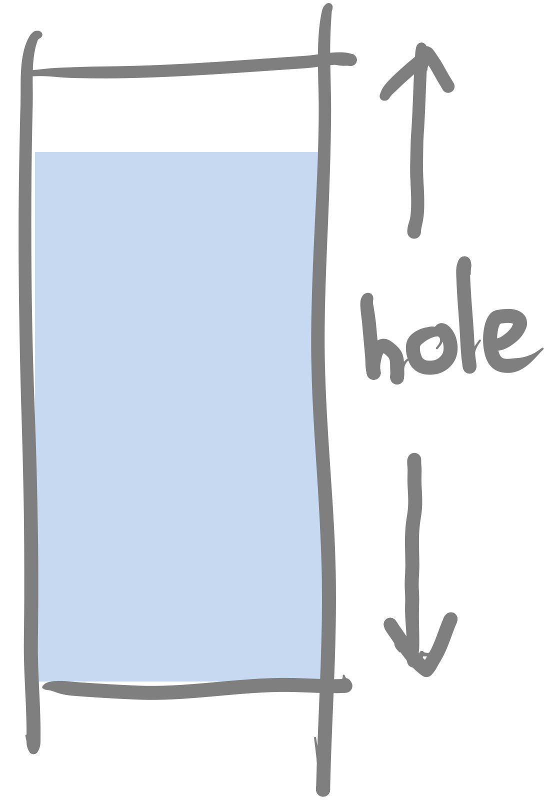


So look for the holes

- Intuition:
  - repeatedly allocate large chunks of memory of size **L** until we find the "right size"

Succeeds!

Sizeof(Hole) ≥ L

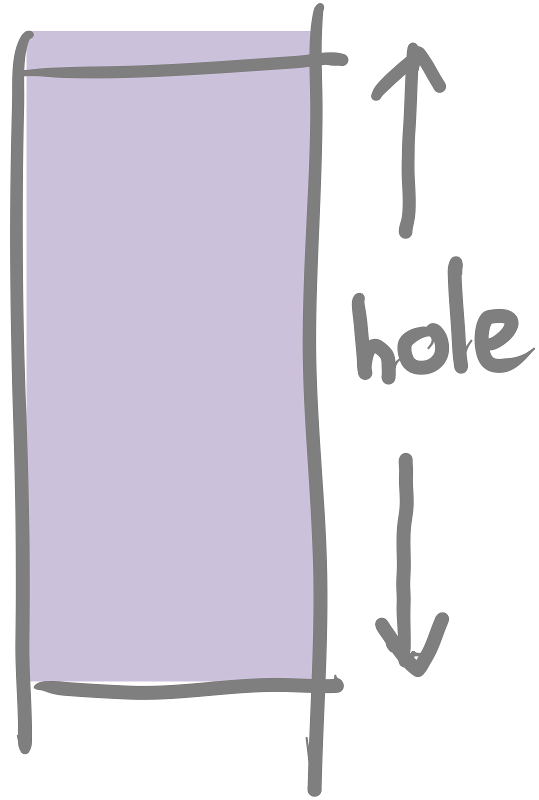


So look for the holes

- Intuition:
  - repeatedly allocate large chunks of memory of size **L** until we find the “right size”

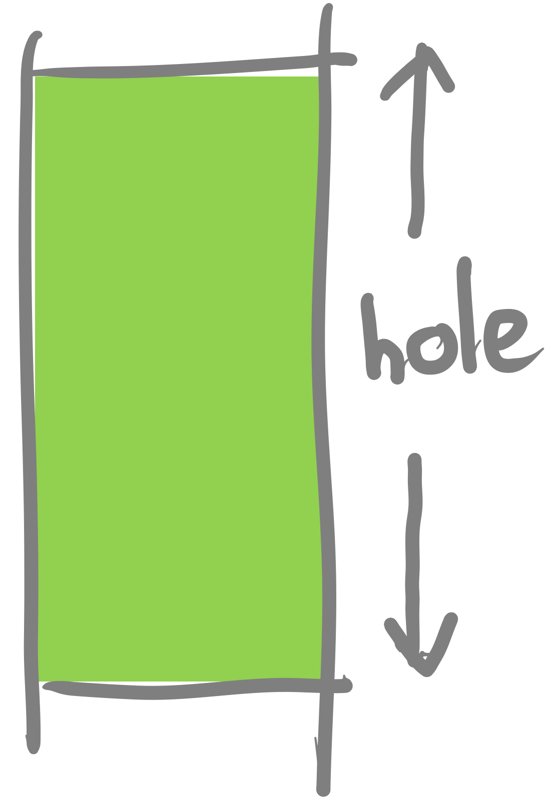
Too large, alloc fails!

Sizeof(Hole) < L



So look for the holes

- Intuition:
  – repeatedly allocate large chunks of memory of size **L** until we find the "right size"



Nailed it!

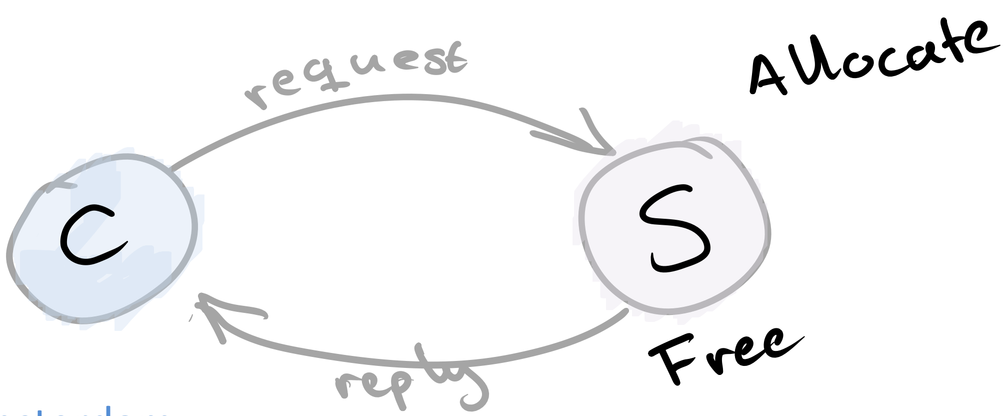
Binary search

Ephemeral Allocation Primitive

- For each probe (i.e., server request):

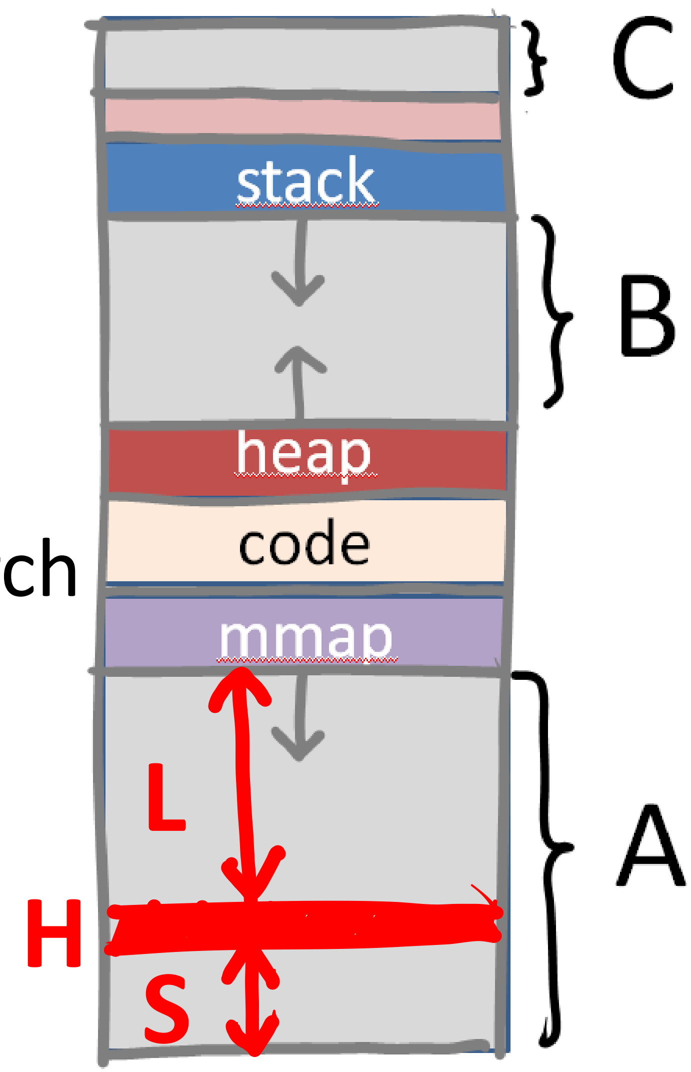
```
ptr = malloc(size);
...
free(ptr);
reply(result);
```

- Strategy: allocation+deallocation, repeat



Ephemeral Allocation Primitive

- Say:
  - Single hidden area is in A (*)
  - Hidden area splits A in two
  - L is the largest hole in AS
  - We can find L via binary search



* See paper for generalization

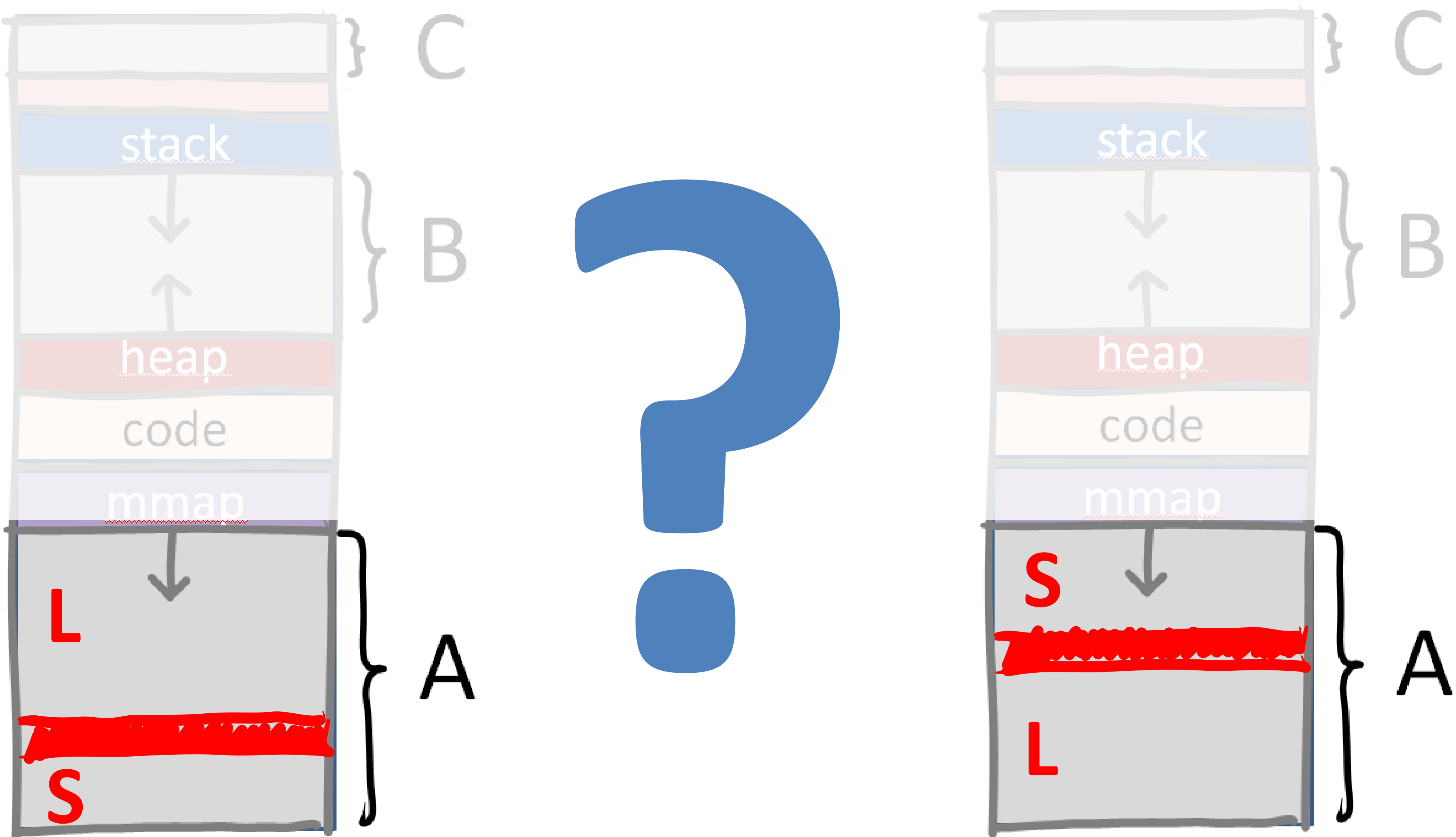
Threat Model

~~Attackers cannot touch the shadow stacks (or any other info that is hidden in this address space)~~

Skip details

Of course we still miss 1 bit of entropy

don't know if large hole is *above* or *below* area



Would be great

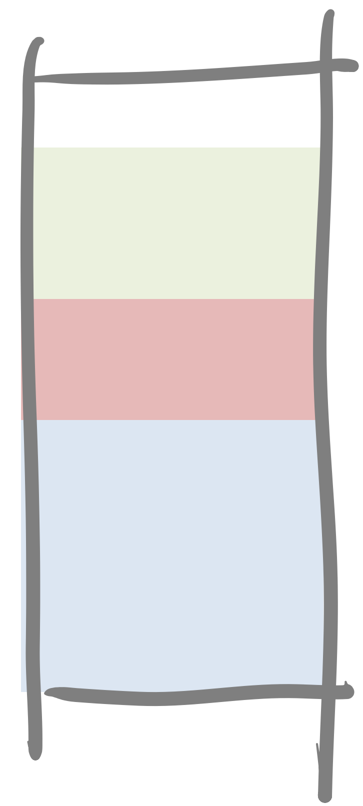
if we could solve this

Persistent Allocation Primitive

- For each request:

```
ptr = malloc(size);
...
reply(result);
```

- Pure persistent primitives rare
- But we can often turn *ephemeral* into *persistent*
  - Keep the connection open
  - Do not complete the req-reply

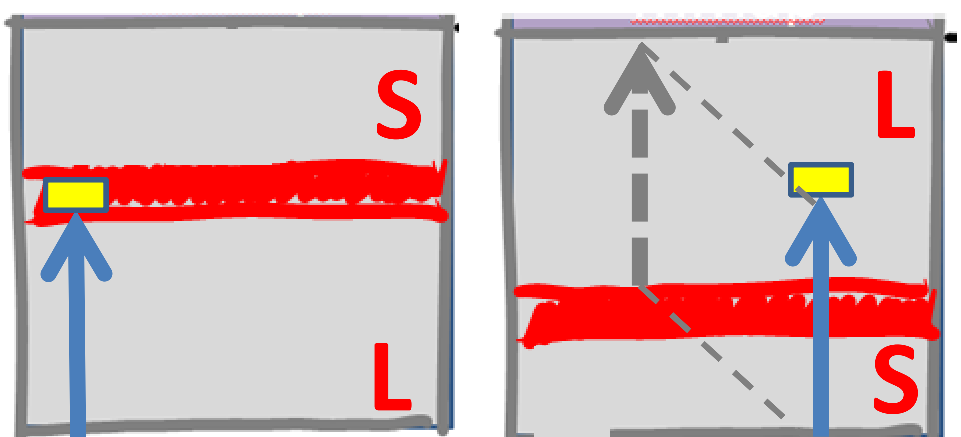


Ephemeral + persistent yields final bit

1. Determine L using ephemeral (binary search)
2. Allocate L using persistent (removing L from AS)
3. Reliably read memory at:

hole_bottom_addr + L

and find either hidden area or 0s:



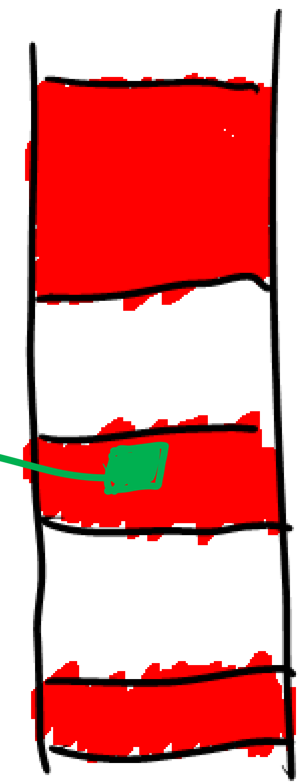
So we need

- A way to effect large allocations repeatedly
- A way to detect whether they failed

Here is what we do

- A way to effect large allocations repeatedly
- A way to detect whether they failed

```
ngx_event_accept(ngx_event_t *ev) {
    ...
    ngx_connection_t *lc = ev->data;
    ngx_listening_t *ls = cl->listening;
    ...
    c->pool = ngx_create_pool(ls->pool_size, ev->log);
    ...
}
```



- When server is in quiescent state
  - Taint all memory
  - See which bytes end up in allocation size

Here is what we do

- A way to effect large allocations repeatedly
- A way to detect whether they failed

Options

- Direct observation (most common)
  - E.g., HTTP **200** vs. **500**
- Fault side channels
  - E.g., HTTP **200** vs. **crash**
- Timing side channels
  - E.g., VMA cache **hit** vs. **miss**

Examples

- Nginx
  - Failed allocation: Connection close.
- Lighttpd
  - We crash both when
    - allocation fails (too large) and
    - succeeds (but allocation > than physical memory)
  - But in former case: crash immediately
  - In latter case, many page faults, takes a long time

Discovered primitives

Program	#	Ephemeral	Persistent	Crash-free
bind	2	✔	✔	✔
lighttpd	3	✔	✔	✗
mysql	3	✔	✔	✔
nginx	5	✔	✔	✔

How fast is it?

- Pretty fast
  - Allocations/deallocations are cheap
  - End-to-end attack is O(log[sizeof(AS)])
  - 37 probes in the worst case on nginx
  - Crash-free, completes in a few seconds
- Existing memory scanning primitives
  - Remote side channels, CROP, etc.
  - End-to-end attack is O(sizeof(AS))
  - 2^{35} probes in the worst case

Practical tips

- How to compile for better security?
  - Focus: clang/llvm
  - Other compilers have similar features (although perhaps not all)
- Different compilation phases
  - Preprocessor
  - Compiler
  - Linker

Basics

debug: Do not pass the -g flag to the compiler, or if you forget, to pass the -Wl,--strip-debug flag to the linker.

strip: Either call strip on the final binary, or pass the -Wl, --strip-all flag to the linker to strip all symbols.

Prepocessor: Fortify Source

Pass the -D_FORTIFY_SOURCE=2 flag to the preprocessor to add extra checks

```
int main(int argc, char *argv[])
{
  char buffer[8];
  strcpy(buffer, argv[0]);
  puts(buffer);
  return 0;
}
```

Since it can compute the size of the buffer at compile time, strcpy will be replaced by strcpy_chk which takes the size of the buffer as a third parameter

ASLR

All new kernels support ASLR. But it is only really meaningful if your code is position independent

-fPIE for executable

-fPIC for libraries

Stack protection

-fstack-protector

- Pass -fstack-protector flag to force addition of a stack canary that checks for stack smashing.
- Use -fstack-protector-strong to include more functions that could be subject to stack smashing
- Use -fstack-protector-all to include all functions (more expensive in code size and execution time).

Safe Stack

- -fsanitize=safe-stack introduces additional stack
  - separated from the unsafe stack,
  - stores return addresses and other pieces of data that may be subject to an attack.

Control Flow Integrity (CFI)

- -fsanitize=cfi flag adds checks that we follow the control flow graph on an indirect call.
  – This requires Link Time Optimization, and so the gold plugin, as activated by -fuse-ld=gold -flto.

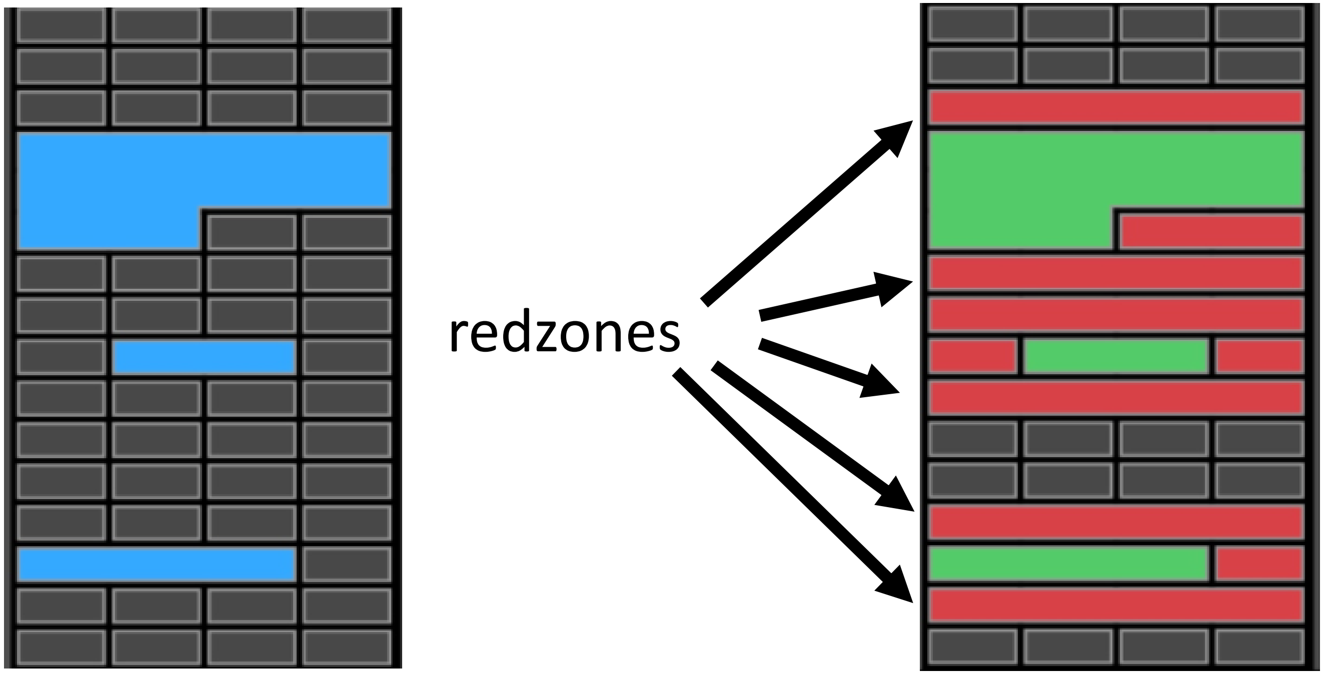
See http://clang.llvm.org/docs/ControlFlowIntegrity.html for more info!

GOT protection

- read-only relocations:
  - Passing -Wl,-z,relro flag to linker marks some section read only, which prevents some GOT overwrite attacks.
- immediate binding:
  - If the -Wl,-z,now flag is passed to the linker, all symbols are resolved at load time. Combined with the previous flag, this prevents more GOT overwrite attacks (otherwise part of the GOT is updated at runtime, and that part is not marked as read-only by -Wl,-z,relro

Address Sanitizer

- Link with -fsanitize=address
- Detects many memory errors: BO, UAF, etc.
- Quite expensive (2x), perhaps only for testing



Undefined behavior

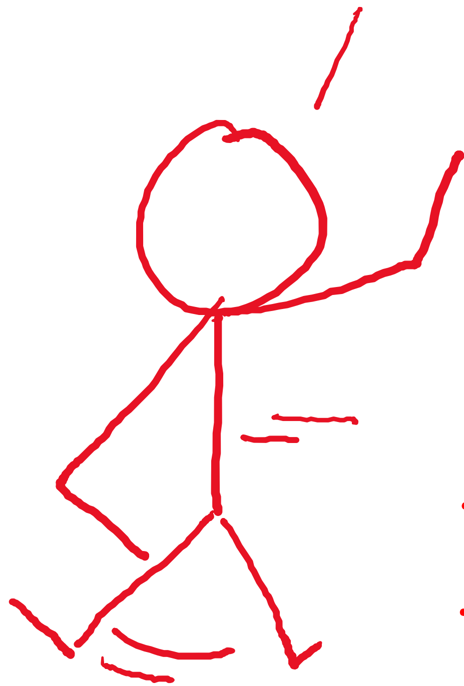
- -fsanitize=undefined
- Undefinged behaviour due to
  - Integer overflow
  - Invalid shifts
  - Etc.

Uninitialized memory

- -fsanitize=memory
- Finds use of unitialized variables

Finally

Let's take a step back



What have these defenses done?

They made it harder to find usable gadgets

The analysis that you need to find them becomes *very* hard

Are we winning?

The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later

Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen[†], Herbert Bos, and Cristiano Giuffrida

VU University Amsterdam

Takeaway

1) Gadget finding: static **dynamic** analysis
2) Compare pretty much **all** defenses

Control-Flow Integrity | Information Hiding | Re-Randomization | Pointer Integrity

3) **Break** them

Static Flesh on the Bone

Shacham at CCS 2007

- `ret2libc` without any function call
- Combine short instruction sequences to build gadgets
- The first systematic formulation of code reuse

Return-Oriented Programming

- Highly influential (900 citations)

CCS 2017 Test of Time Award

Static Flesh on the Bone

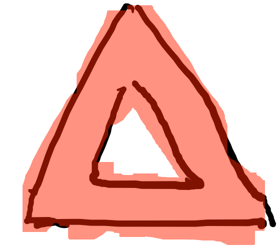
Impact

- Shaped how we think about code reuse:
  1. Analyze the ***geometry*** of victim binary code ➔ static
  2. Locate gadgets
  3. Chain gadgets to craft an exploit
- Initiated much research, almost an arms race

Model never changed

Discover gadgets by means of **static** analysis

Threat Model



Baseline

- ASLR + DEP + Shadow stack (no classic ROP)

Attackers

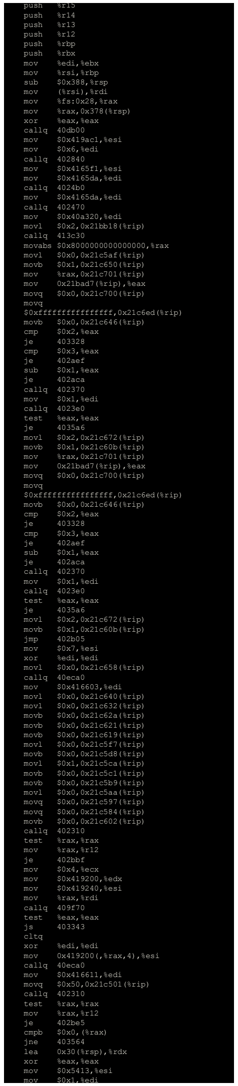
- Arbitrary memory read/write
- Access to the binary
- **Goal is to divert control flow** (no data-only attacks)

Analysis

- Assume <u>perfect</u> implementation of defenses (no sidechannels)
- Focus on server applications

Code-Reuse Research Loop

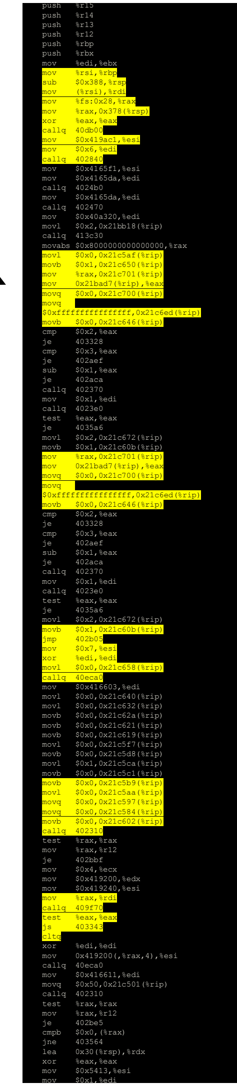
Victim binary code (<u>gadgets</u>)



Attacker

1. Analyze the program

Victim binary code (gadgets)



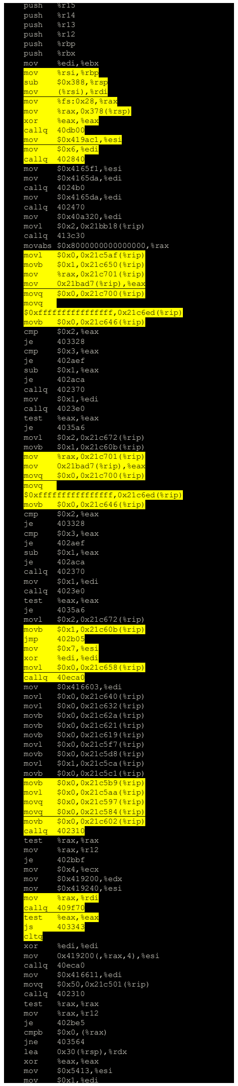
Code-Reuse Research Loop

Attacker

1. Analyze the program
2. Identify gadgets not covered by defenses
3. Publish!

Victim binary code (gadgets)

Code-Reuse Research Loop



Attacker

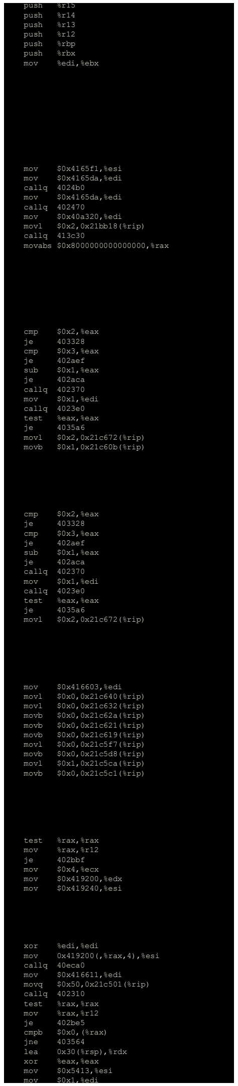
1. Analyze the program
2. Identify gadgets not covered by defenses
3. Publish!

Defender

1. Examine identified gadgets

Code-Reuse Research Loop

Victim binary code (gadgets)



Attacker

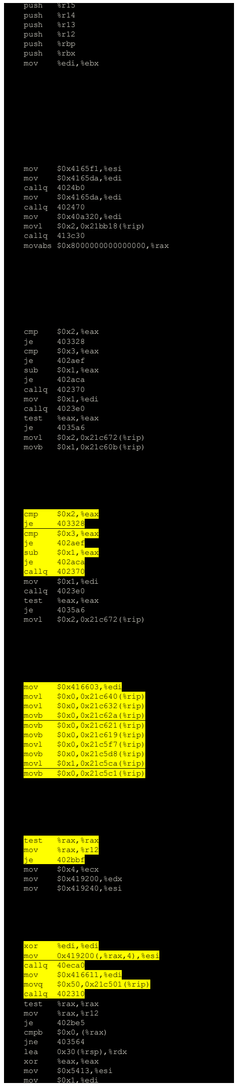
1. Analyze the program
2. Identify gadgets not covered by defenses
3. Publish!

Defender

1. Examine identified gadgets
2. Invent a way to restrict those
3. Publish!

Code-Reuse Research Loop

Victim binary code (gadgets)



Attacker

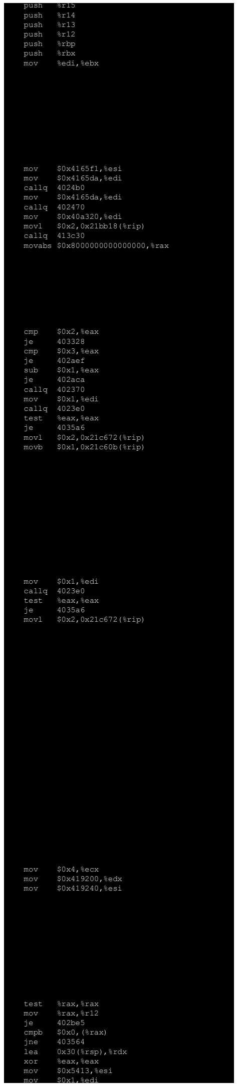
1. Analyze the program
2. Identify gadgets not covered by defenses
3. Publish!

Defender

1. Examine identified gadgets
2. Invent a way to restrict those
3. Publish!

Code-Reuse Research Loop

Victim binary code (gadgets)



Attacker

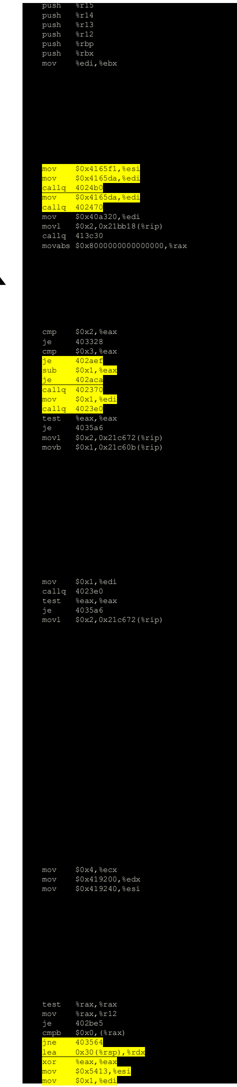
1. Analyze the program
2. Identify gadgets not covered by defenses
3. Publish!

Defender

1. Examine identified gadgets
2. Invent a way to restrict those
3. Publish!

Code-Reuse Research Loop

Victim binary code (gadgets)



Attacker

1. Analyze the program
2. Identify gadgets not covered by defenses
3. Publish!

Defender

1. Examine identified gadgets
2. Invent a way to restrict those
3. Publish!

Code-Reuse Research Loop

Victim binary code (gadgets)

```
push   %r15
push   %r14
push   %r13
push   %r12
push   %rbp
push   %rbx
mov    %edi,%ebx

mov    $0x40a320,%edi
movl   $0x2,0x21bb18(%rip)
callq  413c30
movabs $0x8000000000000000,%rax

cmp    $0x2,%eax
je     403328
cmp    $0x3,%eax

test   %eax,%eax
je     4035a6
movl   $0x2,0x21c672(%rip)
movb   $0x1,0x21c60b(%rip)

mov    $0x1,%edi
callq  4023e0
test   %eax,%eax
je     4035a6
movl   $0x2,0x21c672(%rip)

mov    $0x4,%ecx
mov    $0x419200,%edx
mov    $0x419240,%esi

test   %rax,%rax
mov    %rax,%r12
je     402be5
cmpb   $0x0,(%rax)
```

Attacker

1. Analyze the program
2. Identify gadgets not covered by defenses
3. Publish!

Defender

1. Examine identified gadgets
2. Invent a way to restrict those
3. Publish!

Mindset of Defenders (Academics)

Assume static analysis for gadget retrieval, we

- Constrain control-flow transfers (CFI)
- (re)Randomize code + data
- Enforce pointer integrity

State of the Art of War

- Not many gadgets left (millions > thousands > dozens)
- Remaining gadgets are hard to find

➔ **Code-reuse attacks are hard**

Mindset of Attackers (real world)

Attackers

- Do not care about gadgets or ROP chains or Turing completeness
- Only need to call **`execve`** or **`mprotect`** with controllable args
- Have no reason to limit themselves to static analysis

Change the Model

What memory values should I modify to gain control?

Model this with Dynamic Taint Analysis

Dynamic Analysis

What memory values should I modify to gain control?

1. Get the destination binary into a **quiescent** state
2. **Taint** attacker-controlled bytes (all of rw memory)
3. Monitor branches – **taint sinks** – that depend on tainted memory

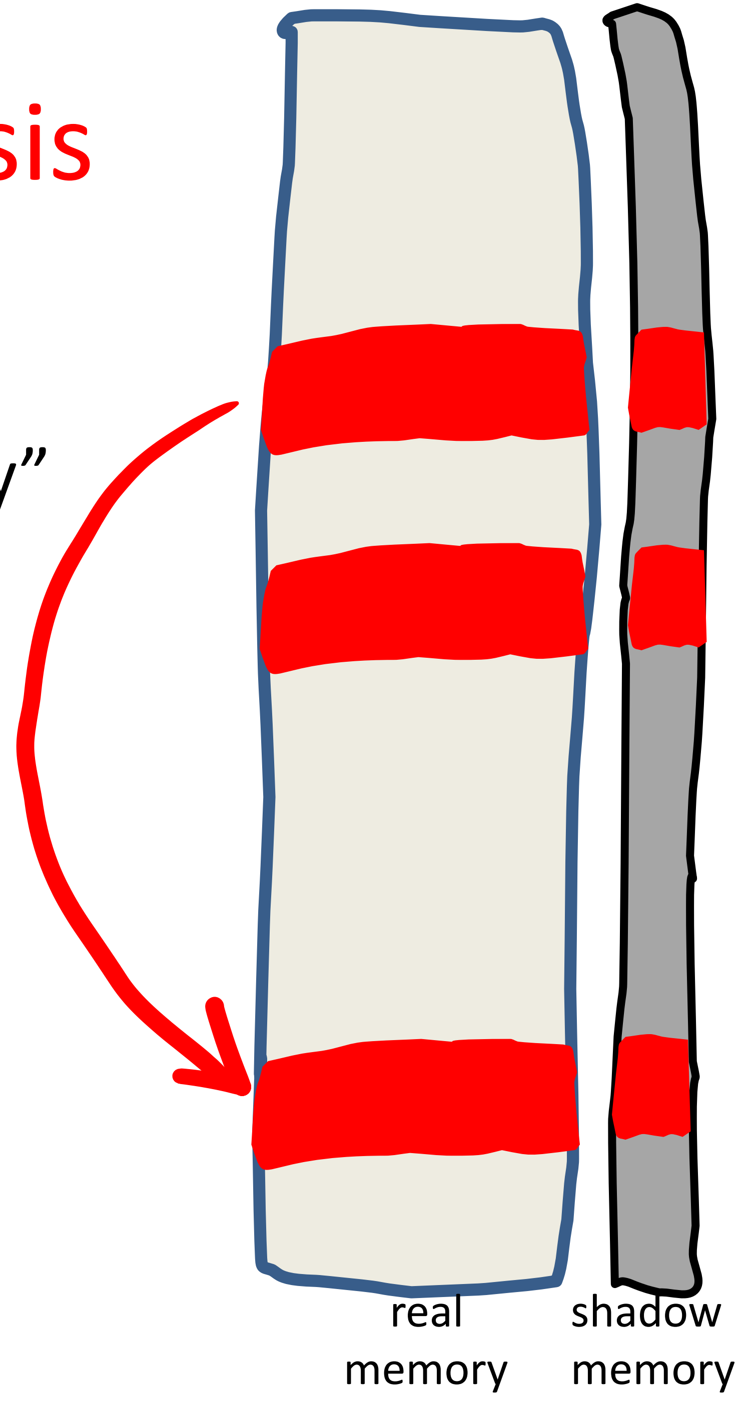
Callsite target + arguments

4. Dump **taint source** for each sink

Taint analysis

- Run in emulator
  - With special "shadow memory"
  - Tracks "taint"
  - Whenever you copy data, you also copy taint

➔ Taint every byte in memory!



Recall the attacker’s game

Two *fundamental* requirements:

- locate code (gadgets)
- jump to it

Defenses

- Recall: defences restrict what attacker may do
- Main restrictions:
  - What values can be modified
  - What code can be targeted

Modeling Code-Reuse Defenses

Write constraints

What memory values can I modify?

Arbitrary memory write: anything in data memory

- Code pointers
- Data pointers
- Other values (integers, characters, …)

A defense may limit what we can corrupt

With Code Pointer Integrity, I cannot modify ***any*** *pointer*

Modeling Code-Reuse Defenses

Target constraints

What can I target?

Arbitrary memory read: any function in code memory

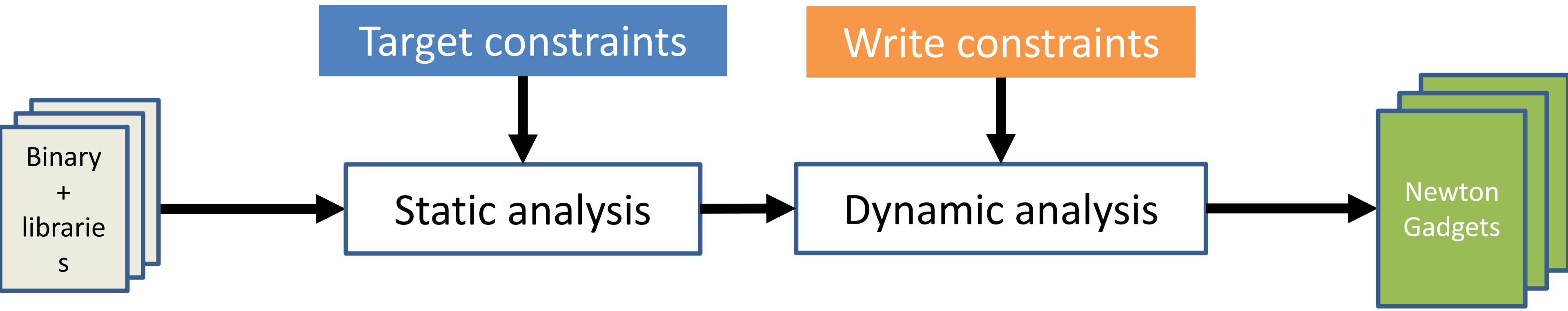
- All functions of the target binary
- All functions of libc
- + any other library

A defense may limit what we can target

With Control-Flow Integrity, I can only target a subset of all functions

Newton

Automated gadget finding with Dynamic Analysis



Newton Gadget

Callsite *cs* is tainted by *addresses* and may call *function*

Newton in Practice (on nginx)

Scenario 1

Baseline

Target constraints

- None – we can target anything

Write constraints

- None – we can corrupt everything

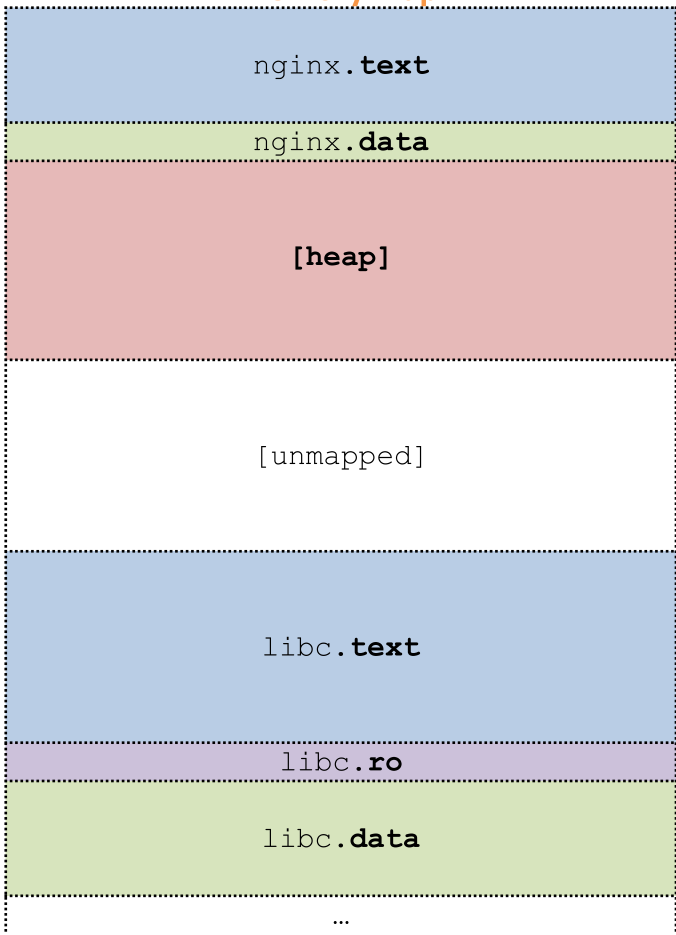
Baseline

$> ./nginx localhost

$> nc -v localhost 80

Connection to localhost 80 port [tcp/http] succeeded!

Memory map



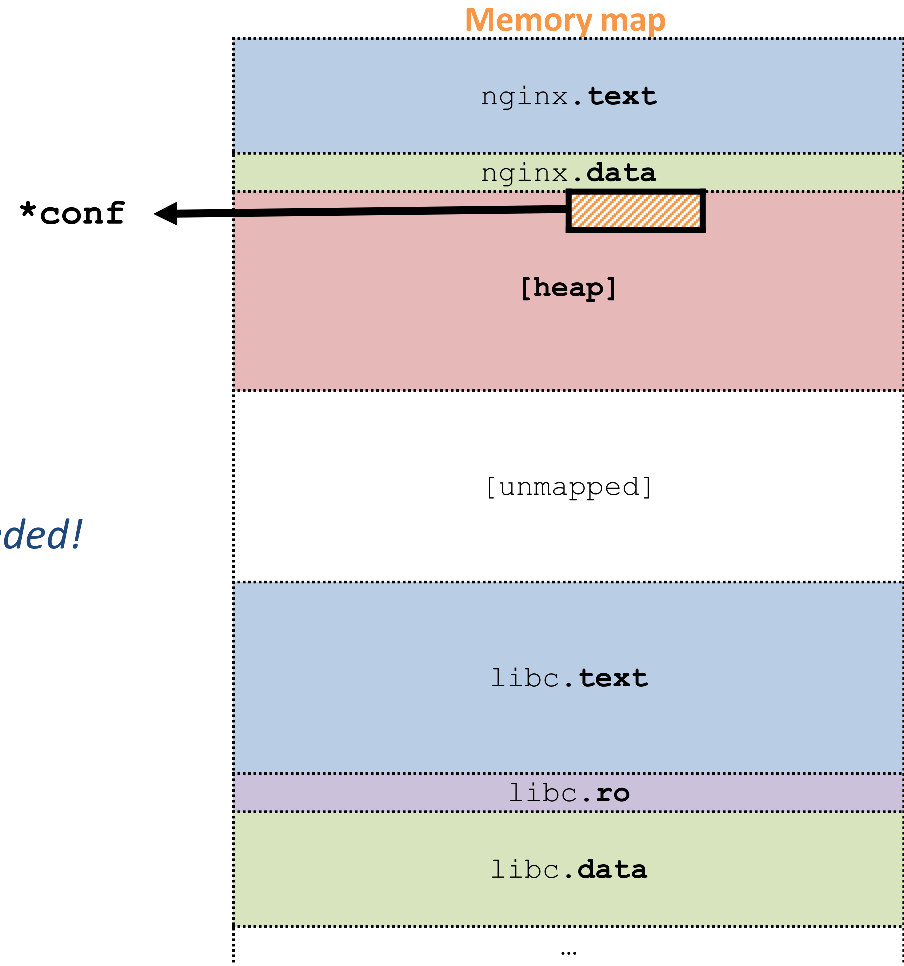
Baseline

$> ./nginx localhost

$> nc -v localhost 80

Connection to localhost 80 port [tcp/http] succeeded!

```
ngx_conf_t *conf;
```



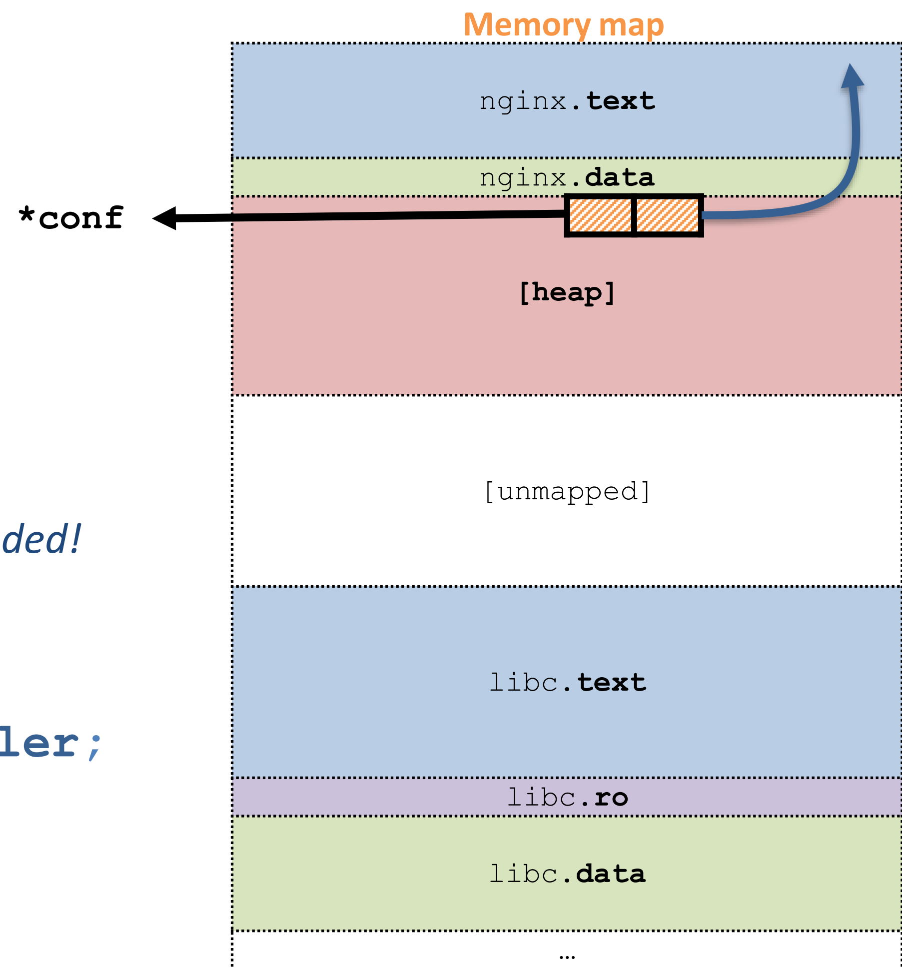
Baseline

$> ./nginx localhost

$> nc -v localhost 80

Connection to localhost 80 port [tcp/http] succeeded!

```
ngx_conf_t *conf;
conf->handler = ngx_proxy_handler;
```



VU University Amsterdam

Quiescent State

- Minimal set of interaction
- Server is stable
- Only long-lived data in memory

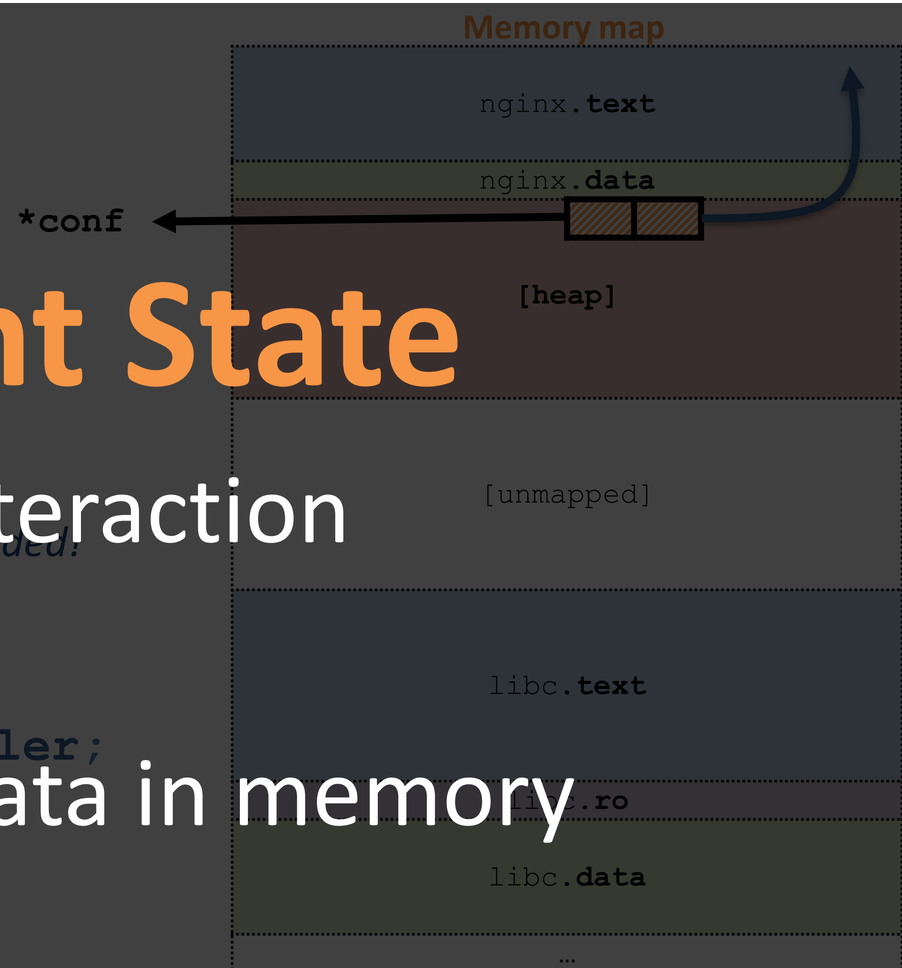
Baseline

$> ./nginx localhost

$> nc -v localhost 80

Connection to localhost 80 port [tcp/http] succeeded!

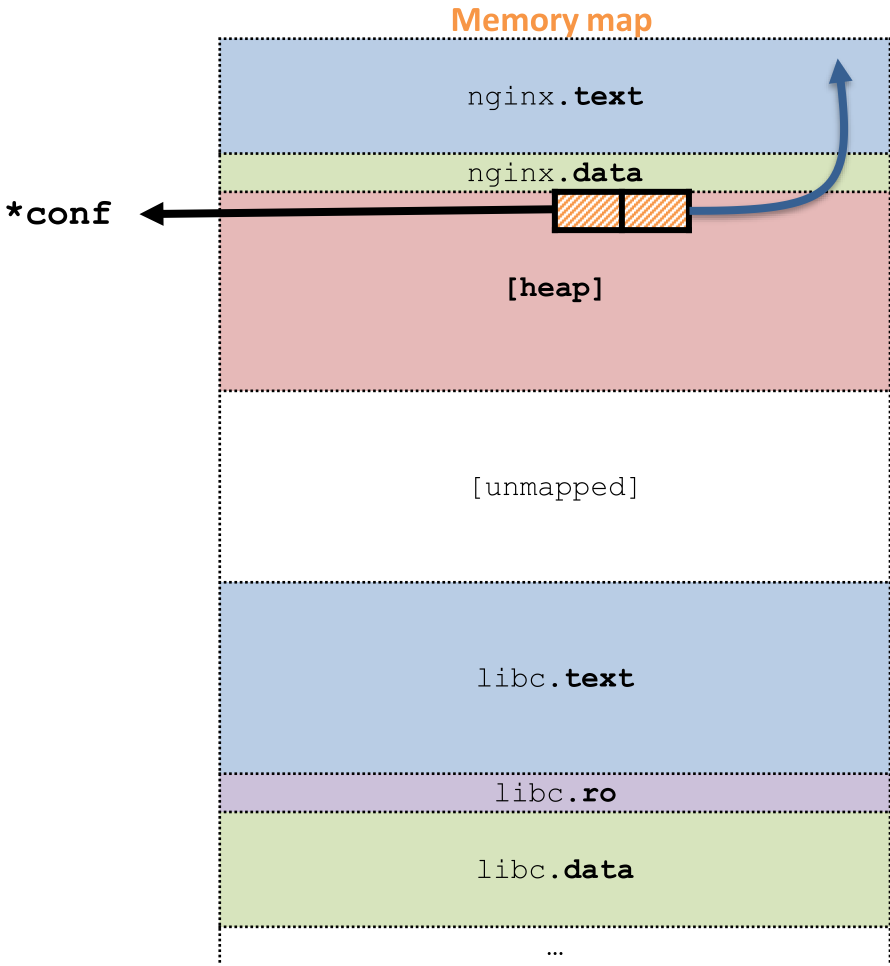
```
ngx_conf_t *conf;
conf->handler = ngx_proxy_handler;
```



Baseline

Quiescent State

[newton] $> taint-all-memory



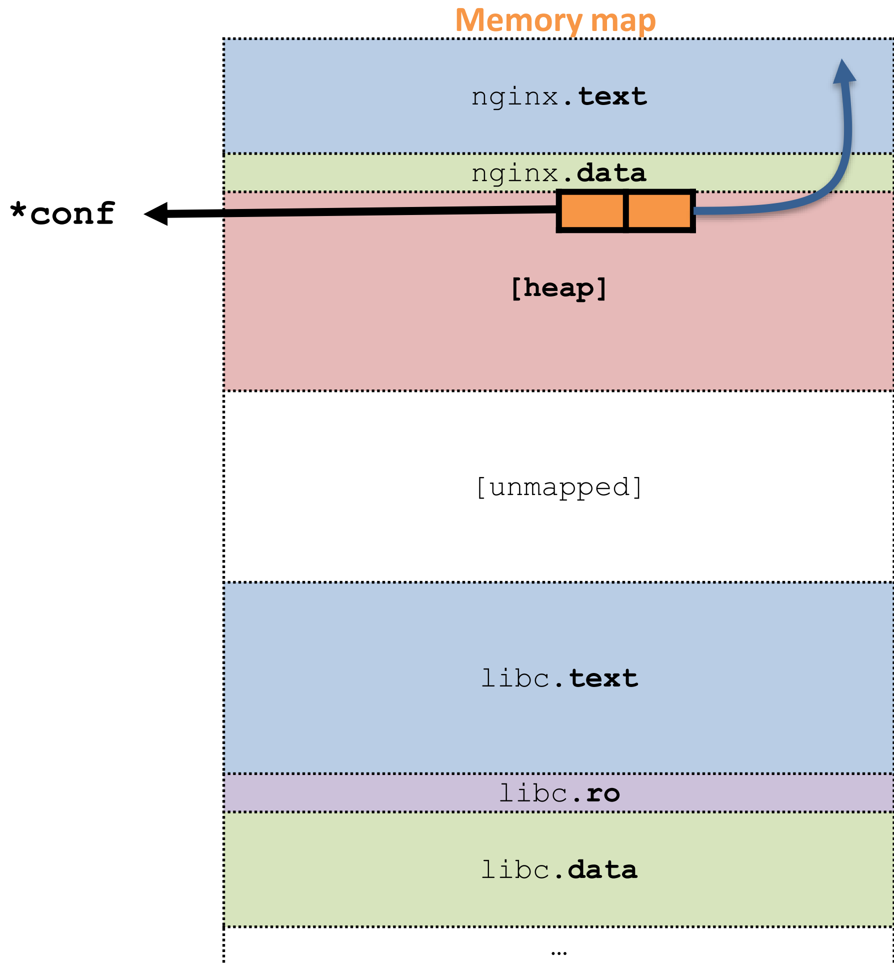
Baseline

Quiescent State

[newton] $> taint-all-memory

[newton] $> monitor-indirect-calls

GET / HTTP/1.0



Baseline

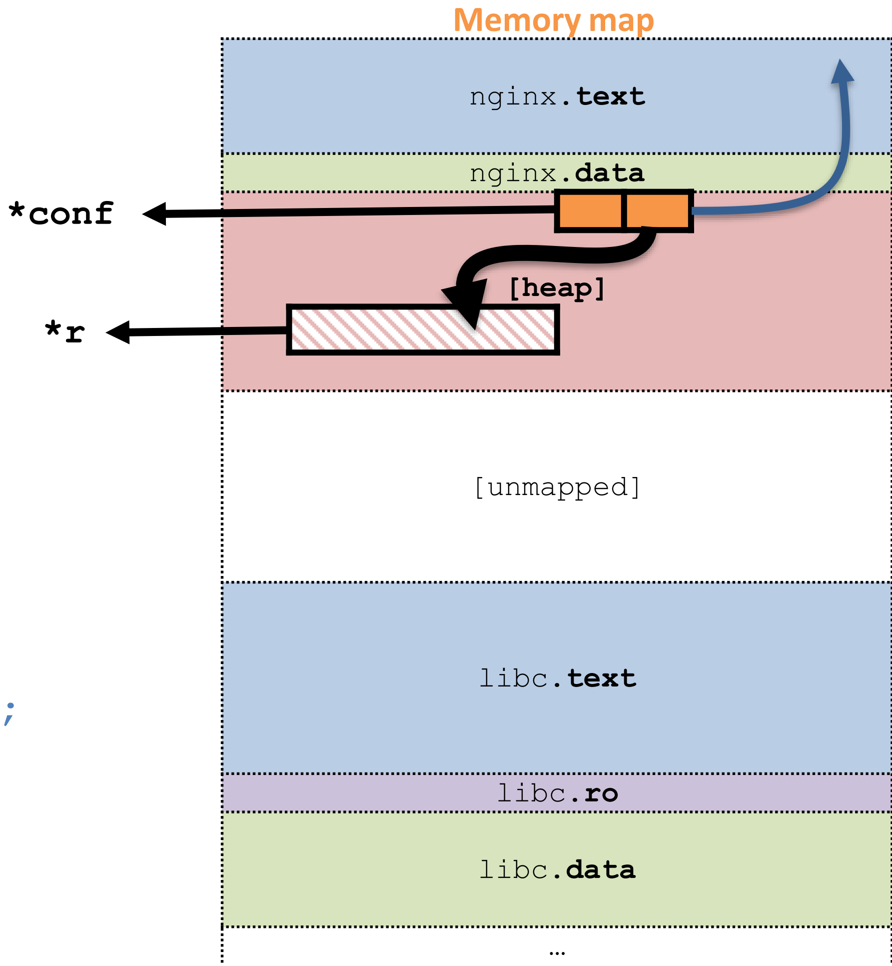
Quiescent State

[newton] $> taint-all-memory

[newton] $> monitor-indirect-calls

```
GET / HTTP/1.0
```

```
ngx_http_request_r *r;
r->content_handler = conf->handler;
```



Baseline

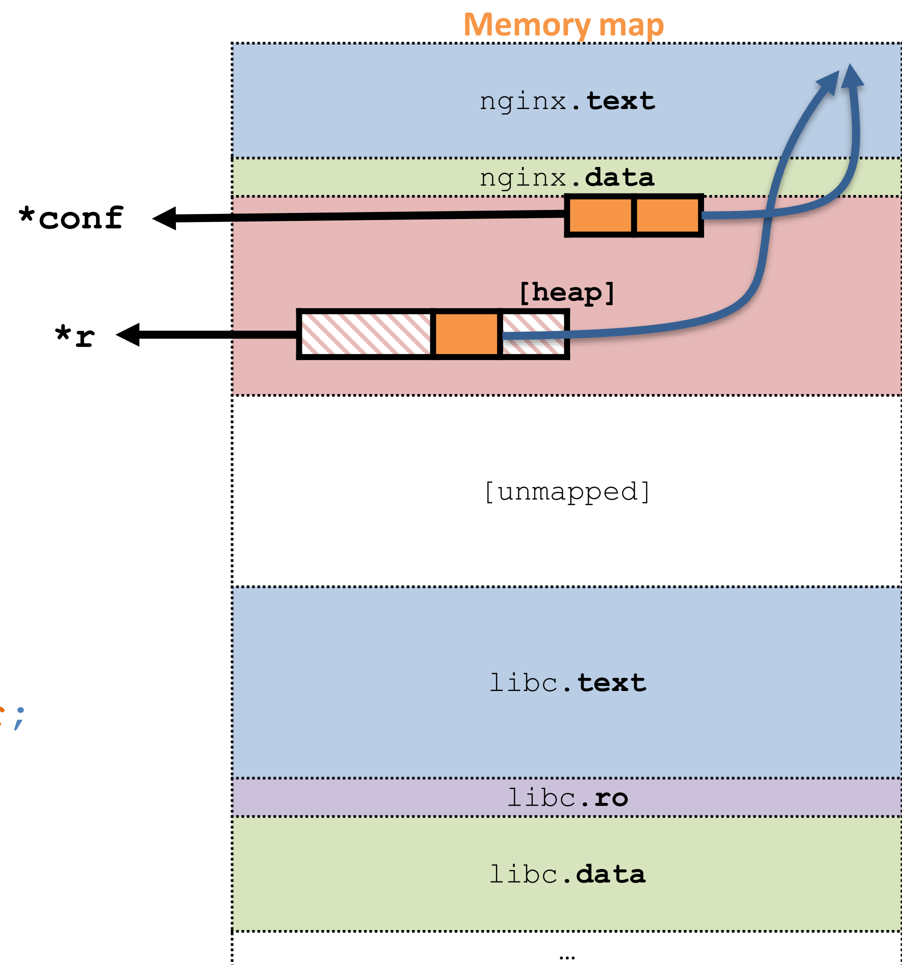
Quiescent State

[newton] $> taint-all-memory

[newton] $> monitor-indirect-calls

```
GET / HTTP/1.0
```

```
ngx_http_request_r *r;
r->content_handler = conf->handler;
r->content_handler(r)
```



Quiescent State

Arbitrary memory read/write

Baseline

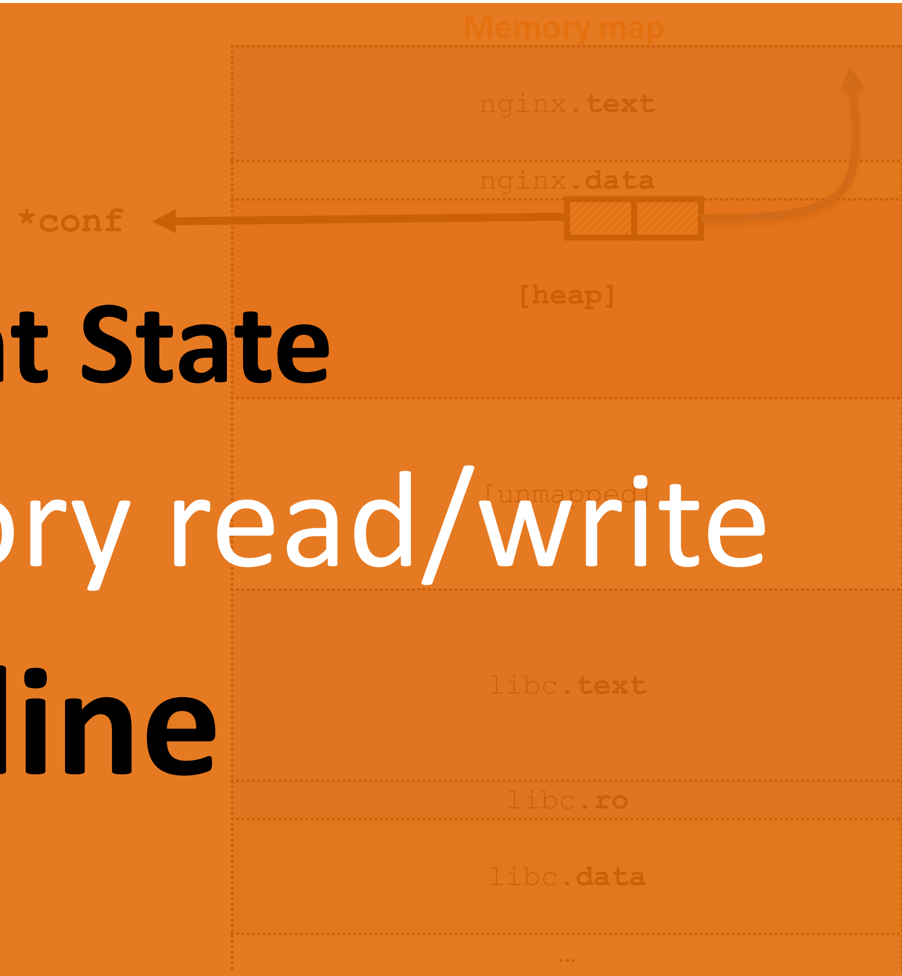
Baseline

$> ./nginx localhost

$> nc -v localhost 80

Connection to localhost 80 port [tcp/http] succeeded!

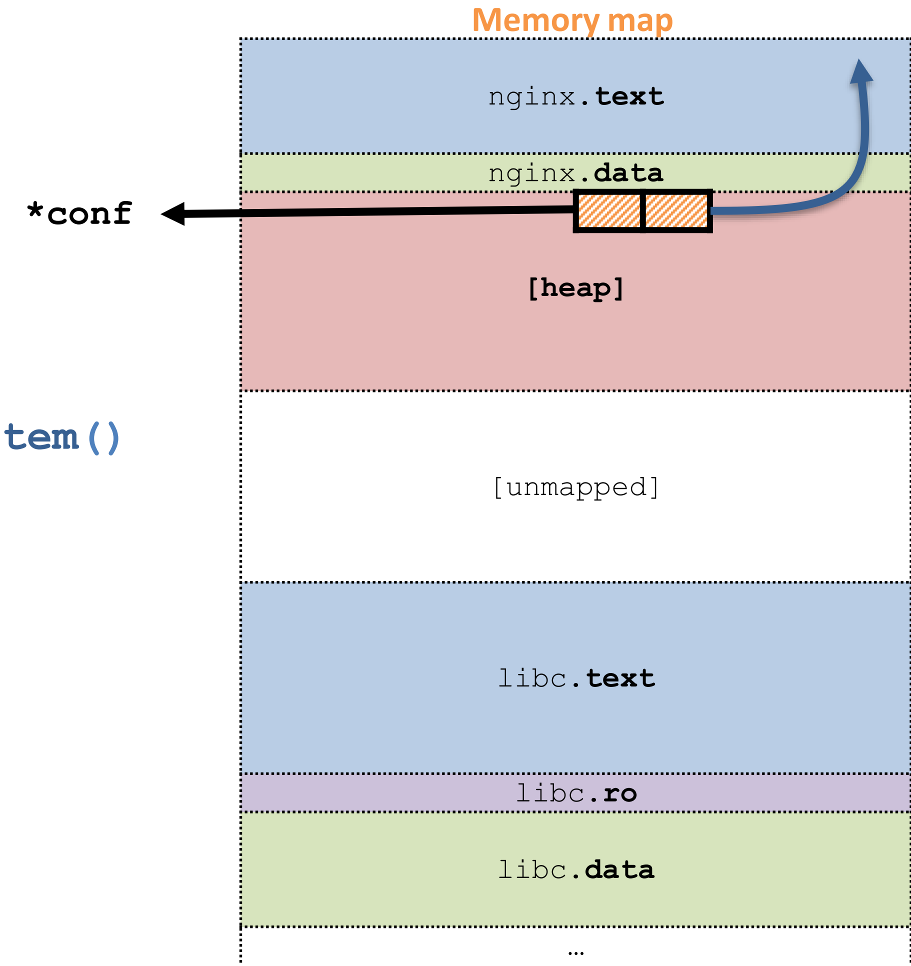
```
ngx_conf_t *conf
conf->handler = ngx_proxy_handler;
```



Baseline

Quiescent State

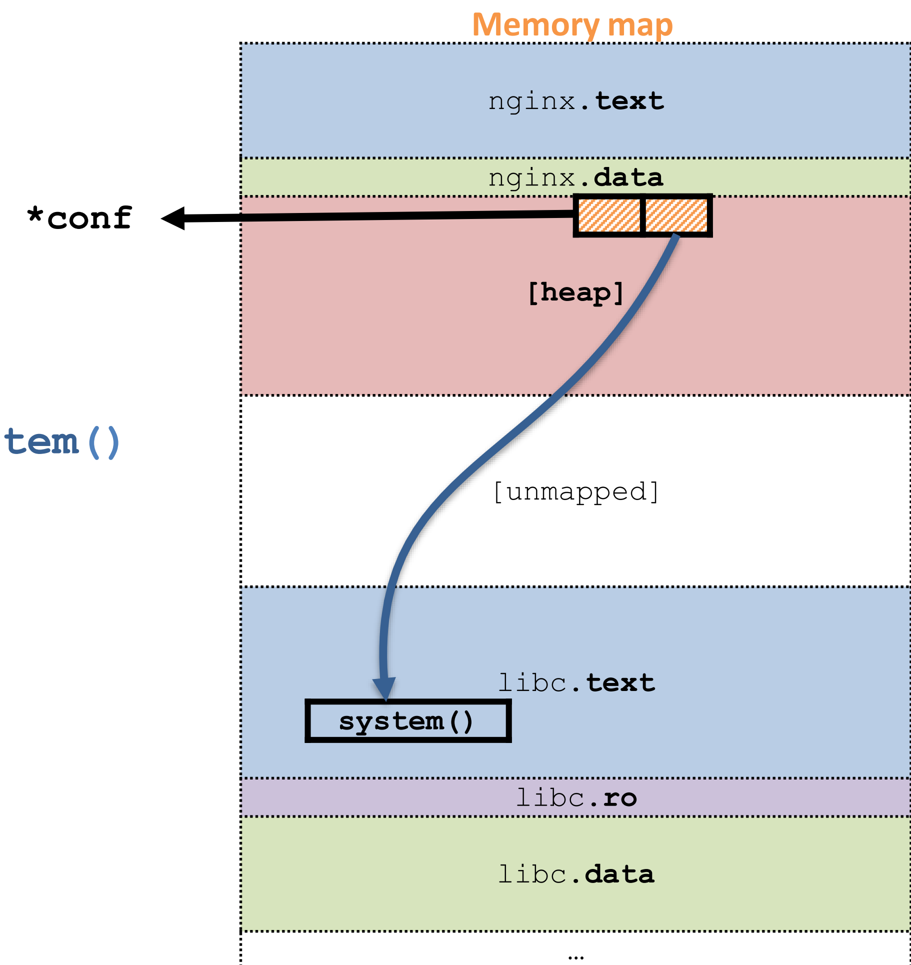
- Let `conf->handler` point to `system()`



Baseline

Quiescent State

- Let `conf->handler` point to `system()`
- Send **GET** request



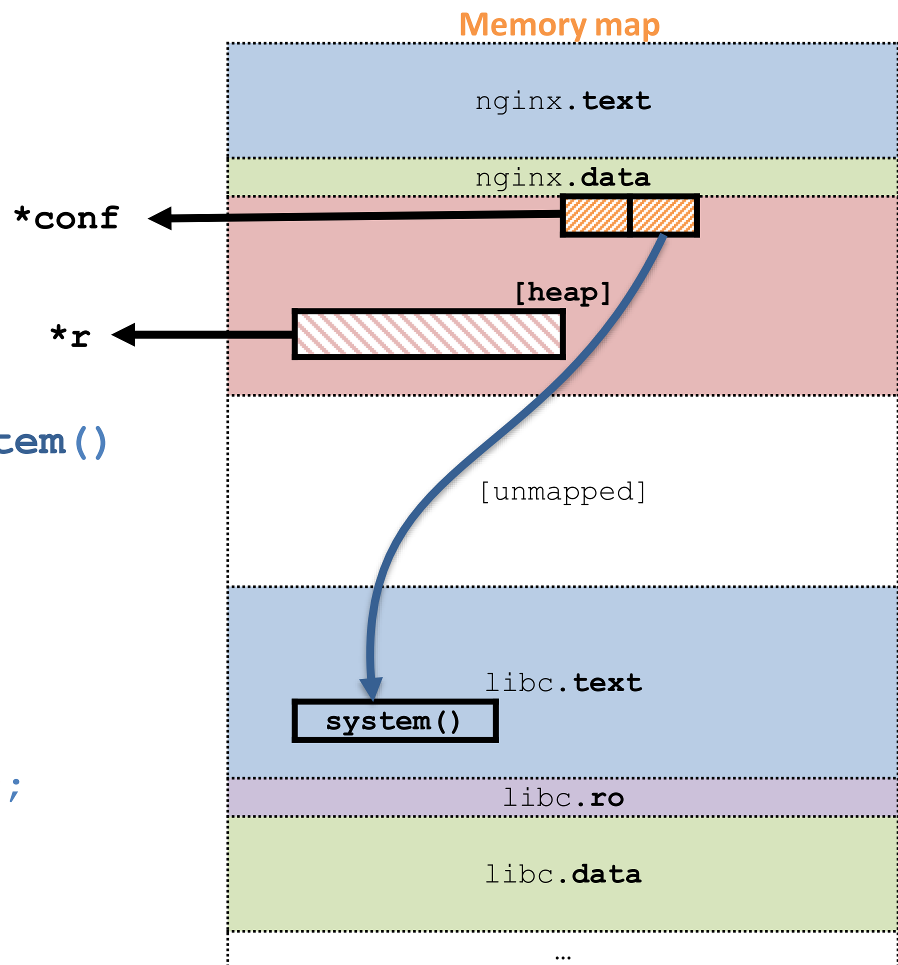
VU University Amsterdam

Baseline

Quiescent State

- Let `conf->handler` point to `system()`
- Send **GET** request

```
ngx_http_request_r *r;
r->content_handler = conf->handler;
```

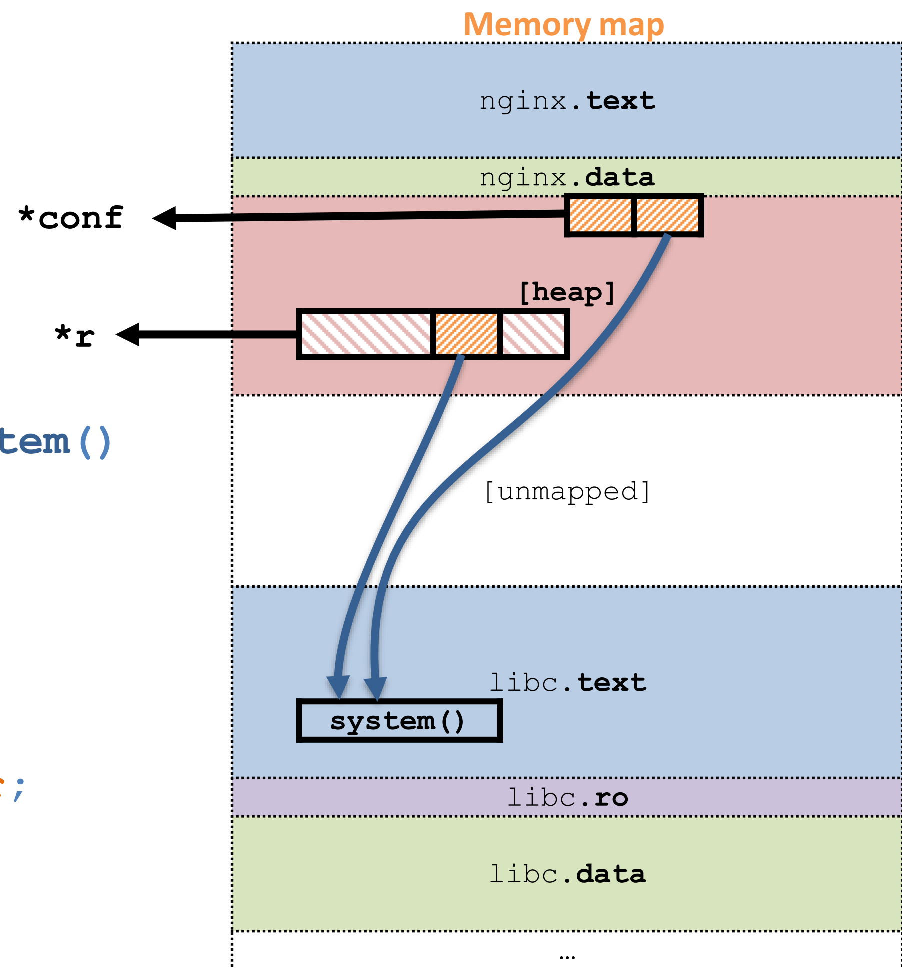


Baseline

Quiescent State

- Let `conf->handler` point to `system()`
- Send **GET** request

```
ngx_http_request_r *r;
r->content_handler = conf->handler;
r->content_handler(r);
```



Newton in Practice (on nginx)

Scenario 1

Baseline

Target constraints

- None – we can target anything

Write constraints

- None – we can corrupt everything

Newton in Practice (on nginx)

Scenario 2

Baseline + XnR + Cryptographic CFI (CCFI)

Target constraints

- No access to code pages
- Only target live code pointers

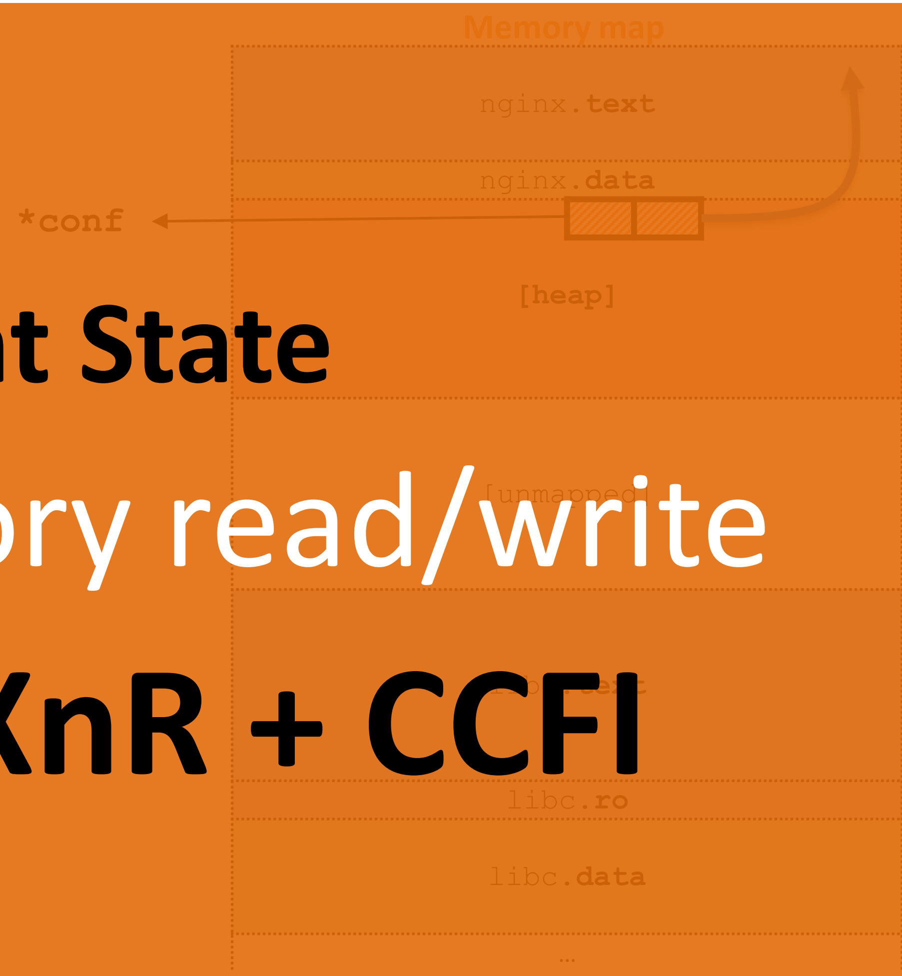
Write constraints

- We can corrupt everything, **except** code ptrs eXecute-not-Read

Quiescent State

Arbitrary memory read/write

Baseline + XnR + CCFI



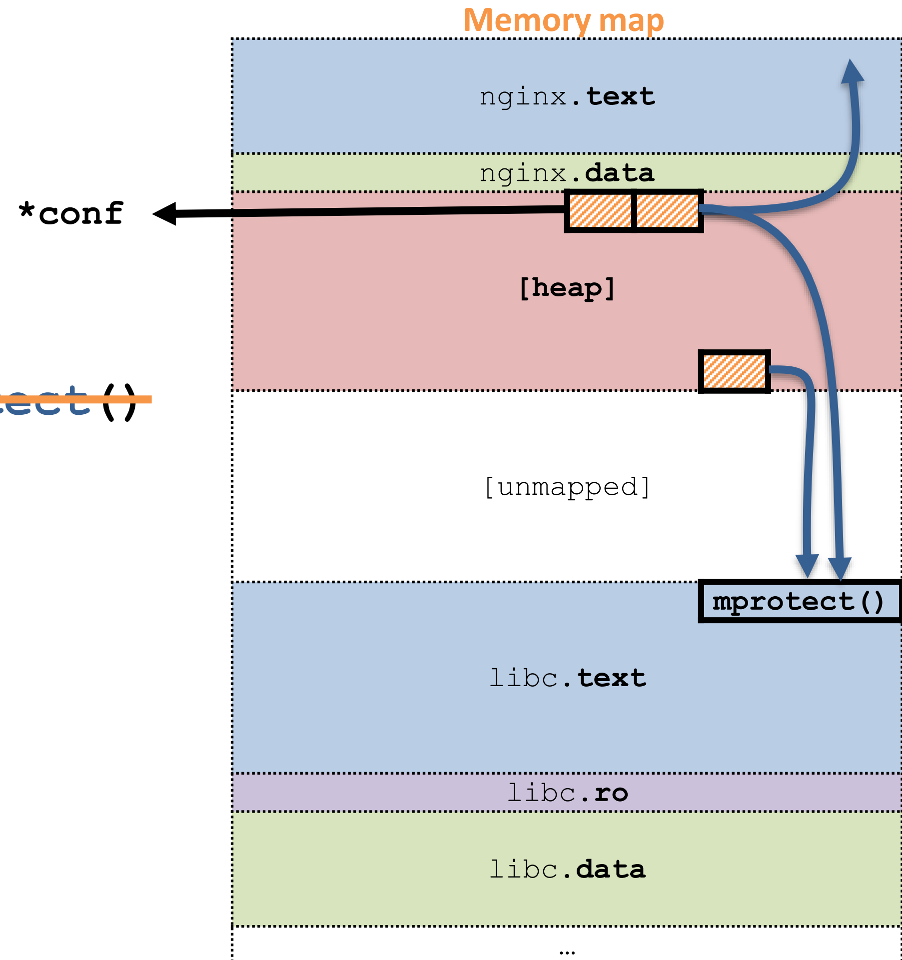
Cryptographic CFI

Quiescent State

- ~~Let conf->handler point to mprotect()~~

CCFI encrypts code pointers

- Cannot corrupt <u>code</u> pointers
- Corrupt <u>data</u> pointers instead



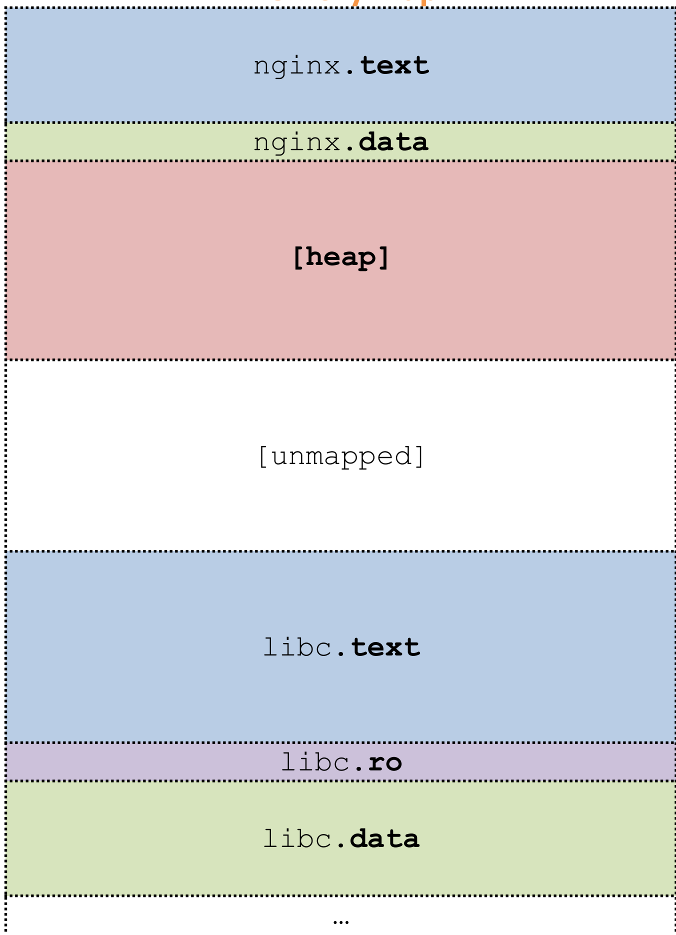
Cryptographic CFI

$> ./nginx localhost

$> nc -v localhost 80

Connection to localhost 80 port [tcp/http] succeeded!

Memory map



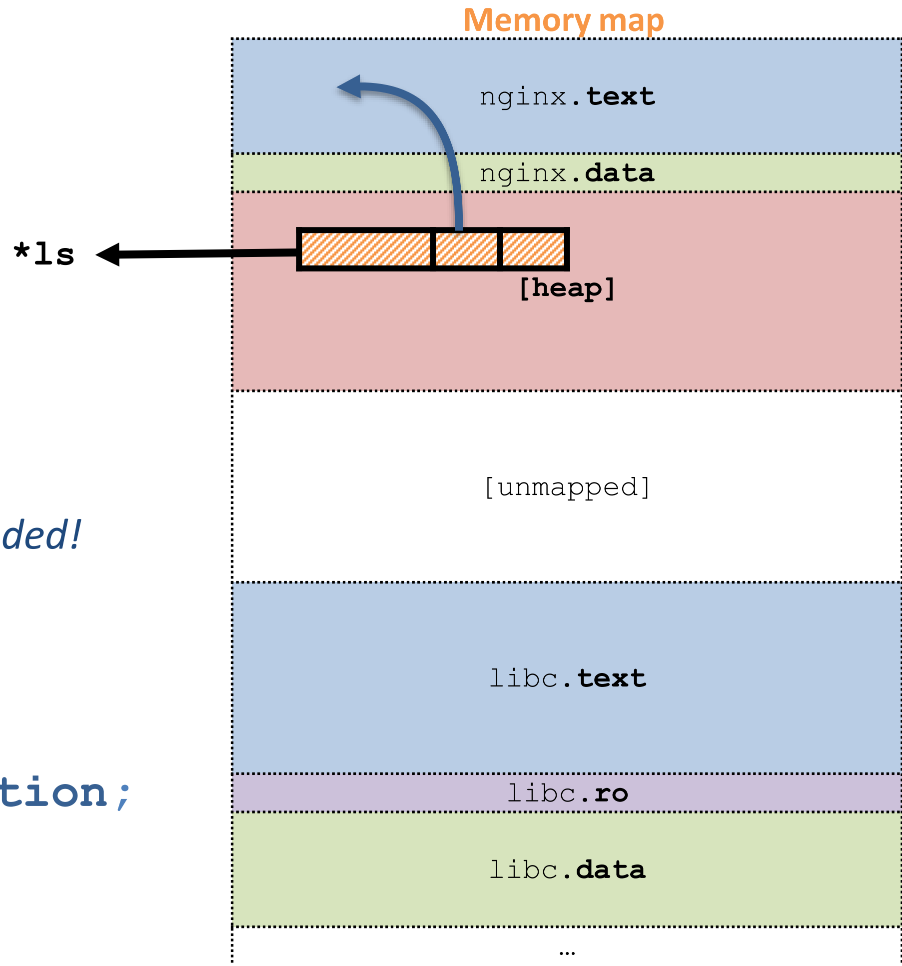
Cryptographic CFI

$> ./nginx localhost

$> nc -v localhost 80

Connection to localhost 80 port [tcp/http] succeeded!

```
ngx_listening_t *ls;
ls->handler = http_init_connection;
```



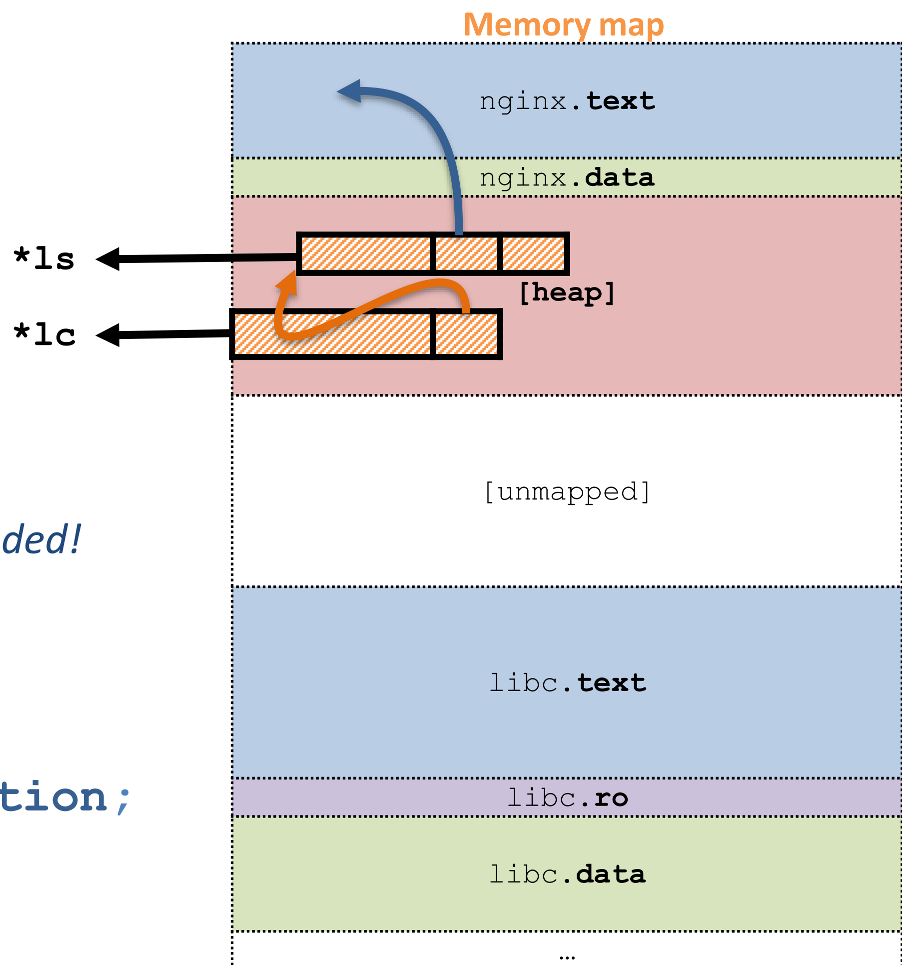
Cryptographic CFI

$> ./nginx localhost

$> nc -v localhost 80

Connection to localhost 80 port [tcp/http] succeeded!

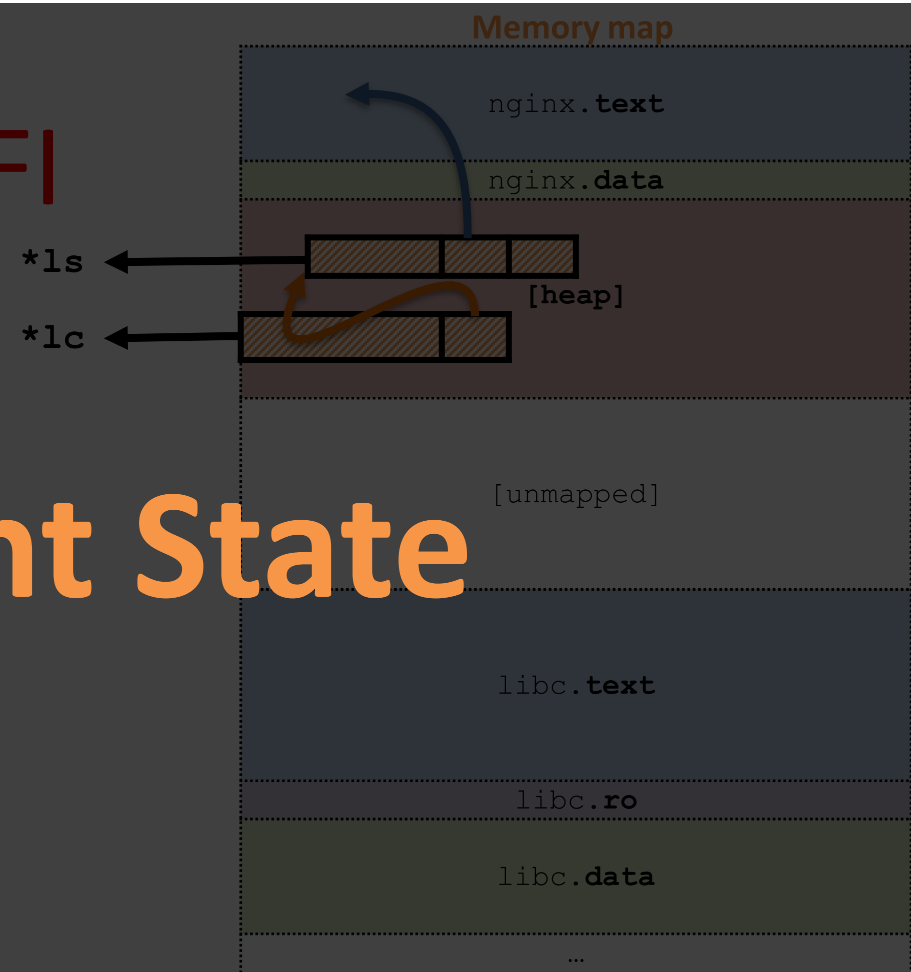
```
ngx_listening_t *ls;
ls->handler = http_init_connection;

ngx_connection_t *lc;
lc->listening = ls
```



Cryptographic CFI

Quiescent State

Quiescent State

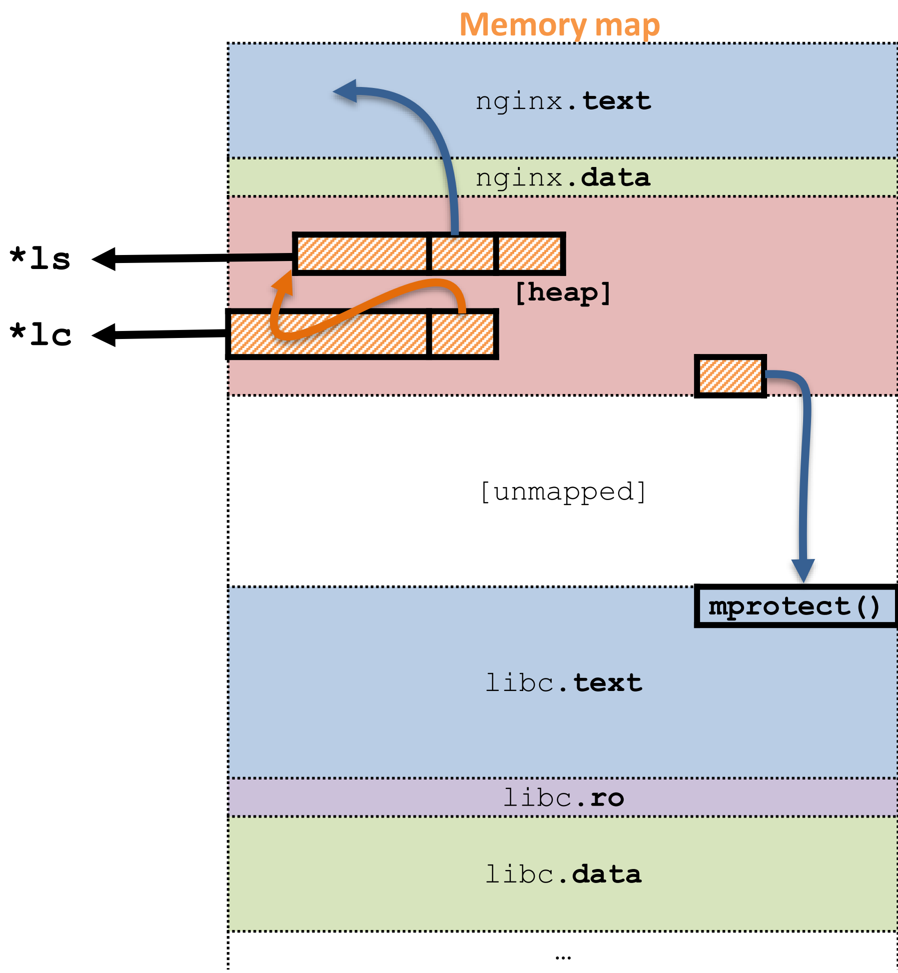


Cryptographic CFI

Quiescent State

[newton] $> get-live-code-pointers

[newton] $> taint-all-memory



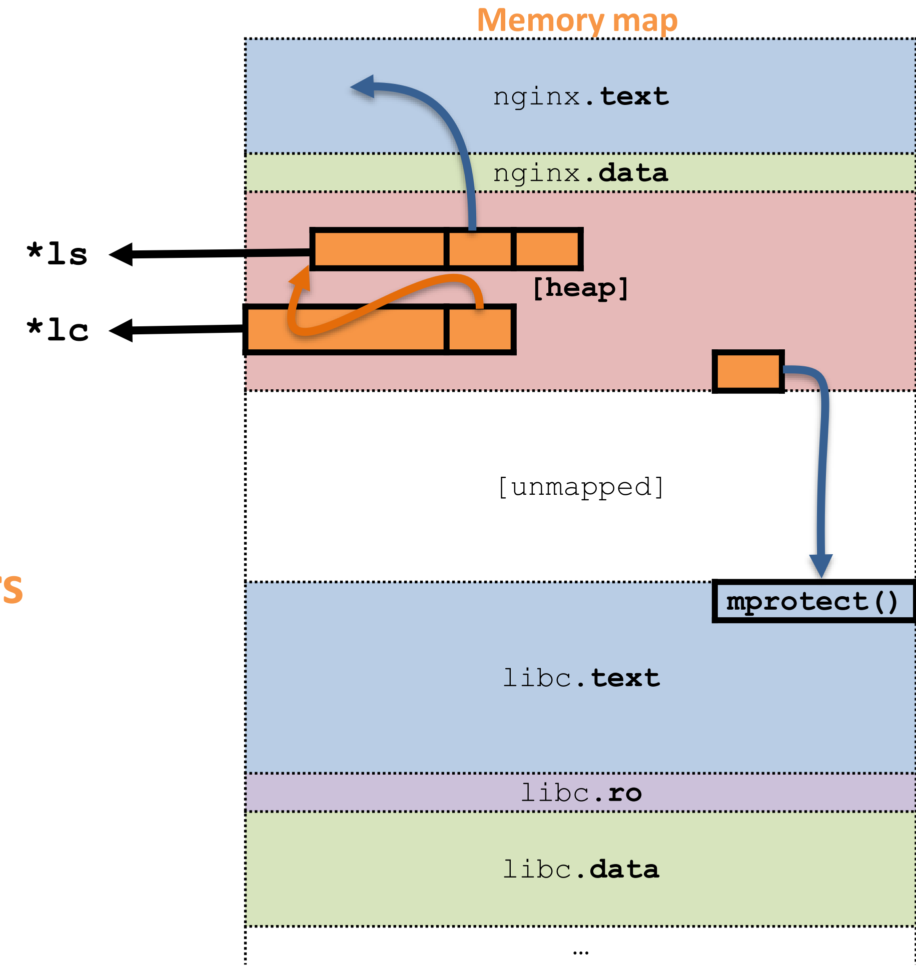
Cryptographic CFI

Quiescent State

[newton] $> get-live-code-pointers

[newton] $> taint-all-memory

[newton] $> taint-wash-code-pointers



Cryptographic CFI

Quiescent State

[newton] $> get-live-code-pointers

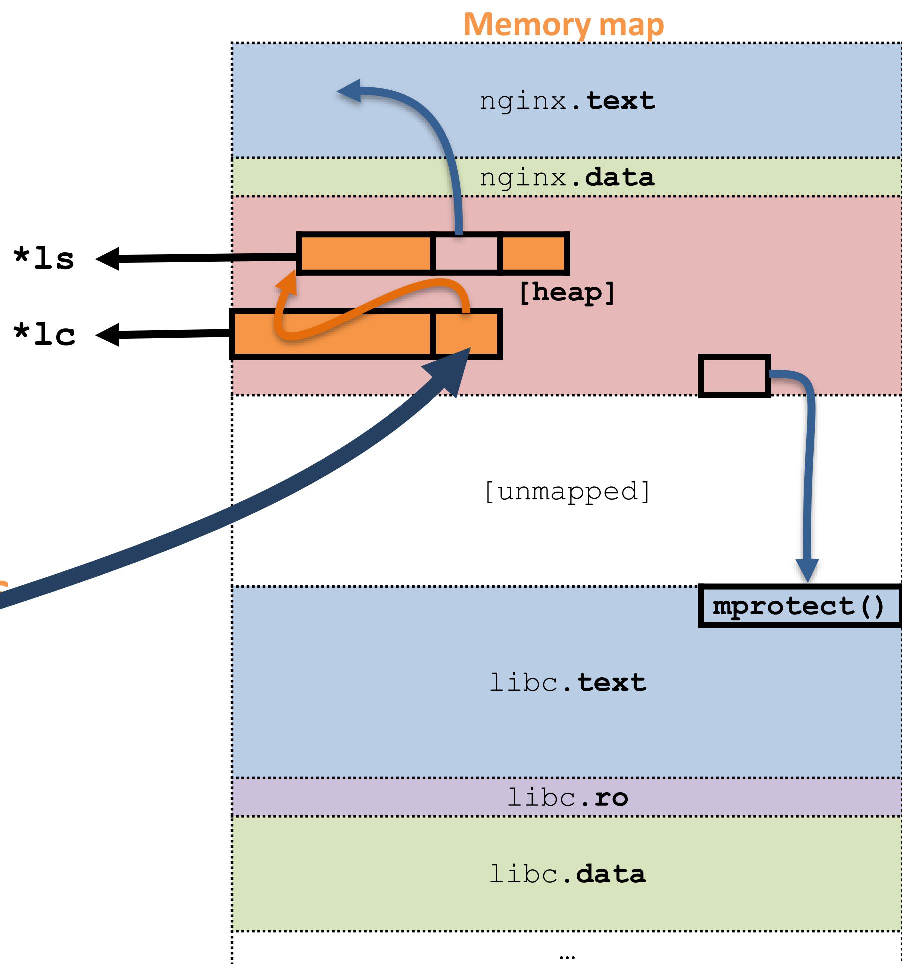
[newton] $> taint-all-memory

[newton] $> taint-wash-code-pointers

[newton] $> monitor-indirect-calls

```
GET / HTTP/1.0
```

```
lc->listening->handler(c);
```



Quiescent State

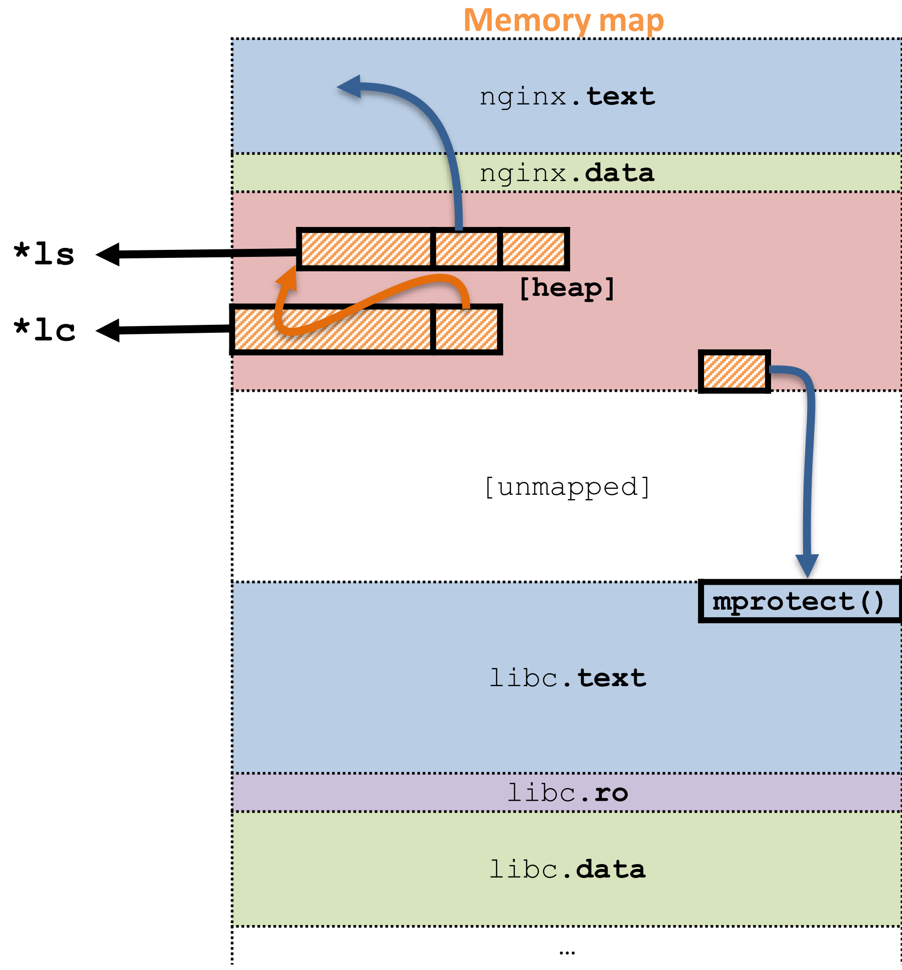
Arbitrary memory read/write

Baseline + XnR + CCFI

Cryptographic CFI

Quiescent State

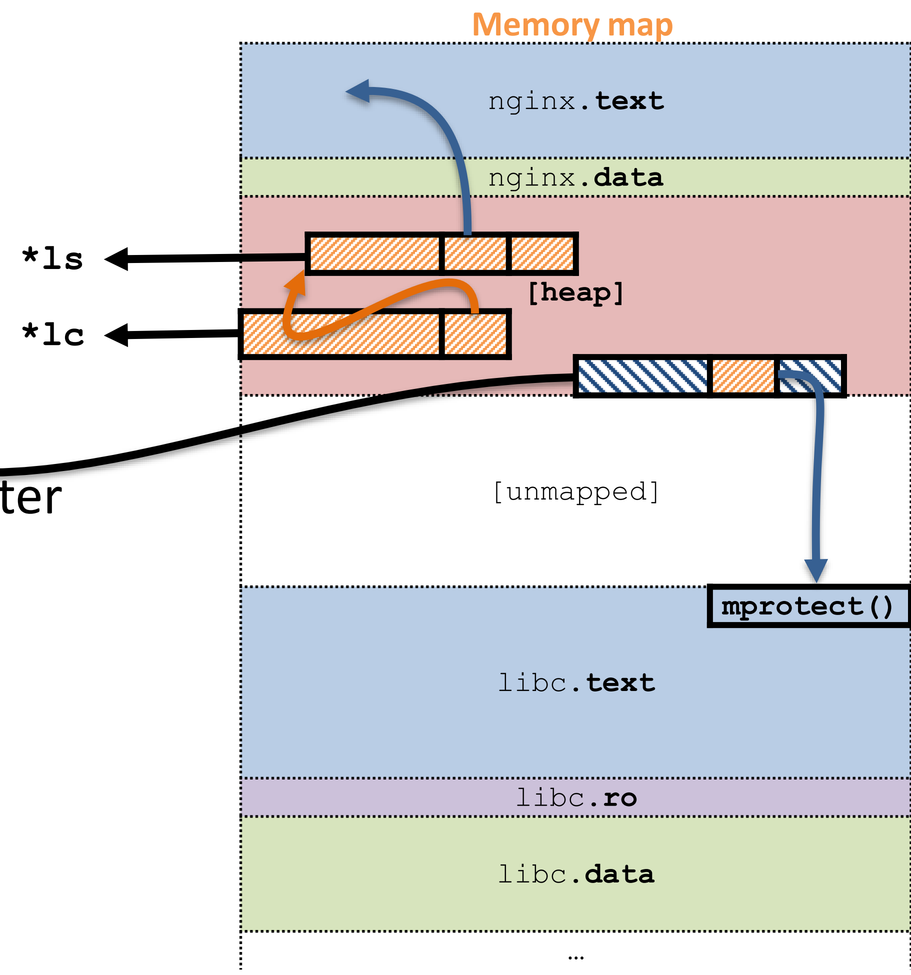
- Construct counterfeit `ls` object



Cryptographic CFI

Quiescent State

- Construct counterfeit `ls` object
- Corrupt `lc->listening` data pointer (make it point to our `ls`)

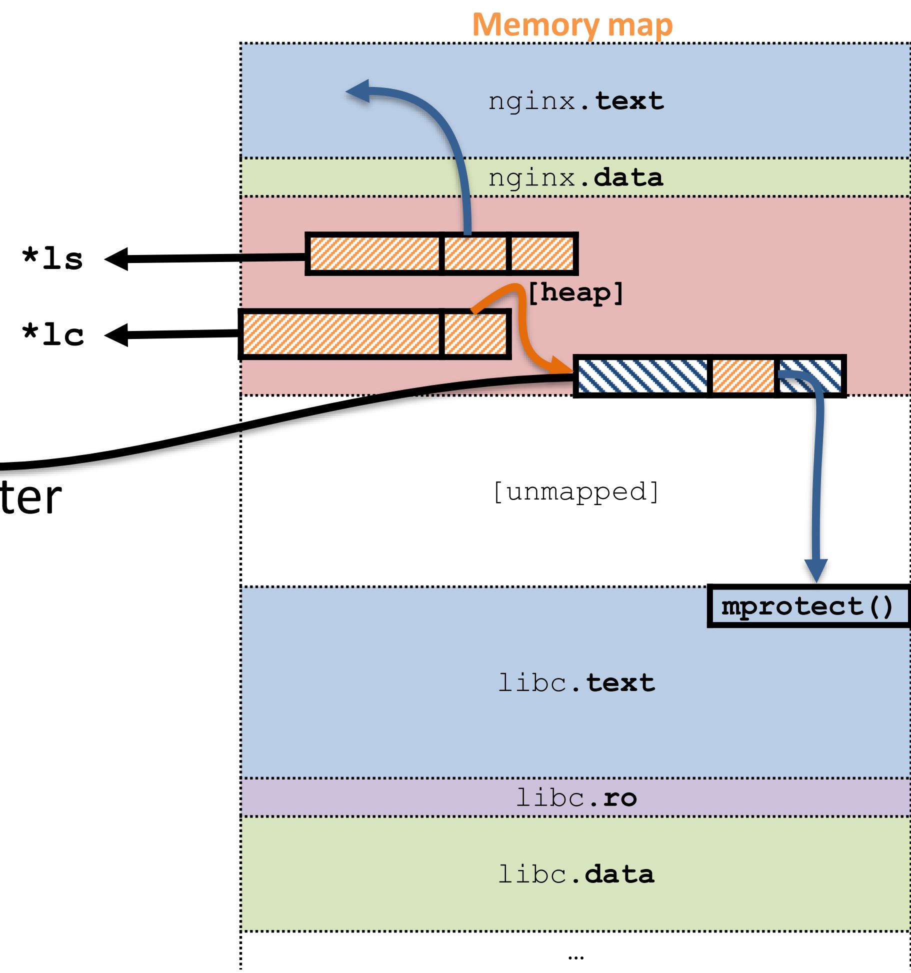


Cryptographic CFI

Quiescent State

- Construct counterfeit `ls` object
- Corrupt `lc->listening` data pointer (make it point to our `ls`)
- Send **GET** request

```
lc->listening->handler(lc);
```



Newton in Practice

Controlling arguments

- Examples only show how to divert control-flow
- Use the same mechanics for <u>arguments</u>

Basically

- It means we got the threat model wrong <u>again</u>

Conclusion

https://vusec.net/newton

10+ years of code-reuse

- Crafting code-reuse attacks is hard
- Attacks and defenses assume static analysis

Newton says

- Consider the dynamics and find that there is still leeway
- Use reported gadgets to compare defenses

The next 10 years of code-reuse

- Combine state-of-the-art defenses to reduce exploitability
- Reduce overhead of more heavyweight defenses

What to do?

Consider UNIX

- Since 1970s
- Many eyes

Desktops	: 8%
Game console	: 30%
Mainframes	: 30% (more as guest)
Embedded	: 35%
Servers	**: 70%**
Tablets	**: > 90%**
Smartphones	**: > 90%**
Supercomputers	**: ~*all***

SROP

Erik Bosman

Need as few as one gadget:

"**syscall** (0x0f05) & **ret** (0xc3)"

- Always present
- Sometimes at fixed location

"Framing Signals", Security & Privacy, 2014

SROP

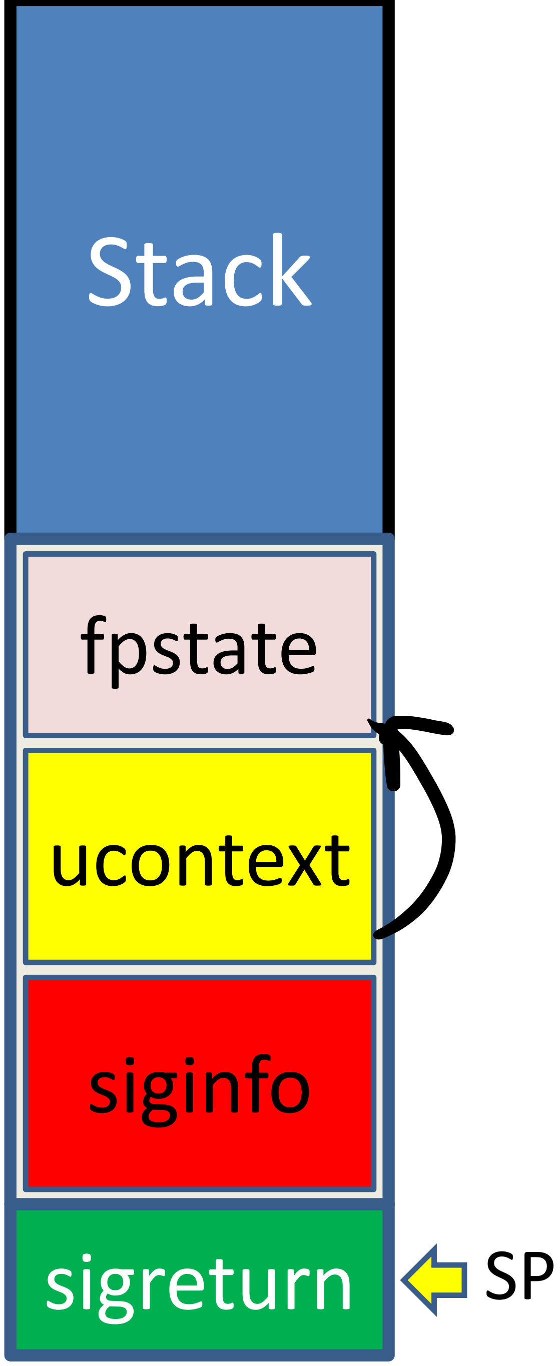
- Abuses UNIX signalling
- Relevant for almost all UNIX systems

SROP

- Make use of a “hidden” system call: sigreturn
- Remember signals?
  - Kernel delivers signal
  - Stops code currently executing
  - Saves context
  - Executes signal handling code
  - Restores original context ➔ sigreturn

UNIX Signal Handling

- Kernel keeps all administration in userland

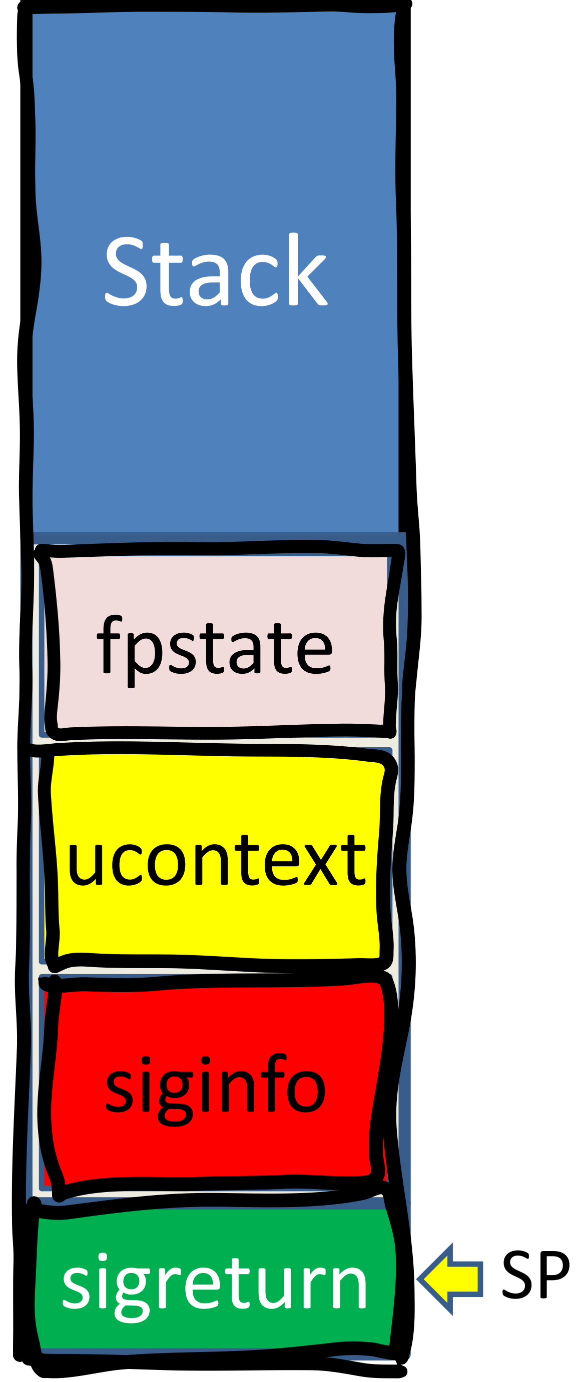


VU University Amsterdam

UNIX Signal Handling

- Kernel keeps all administration in userland
  - Good?
  - Bad?

- Kernel does not remember signals
  - Good?
  - Bad?

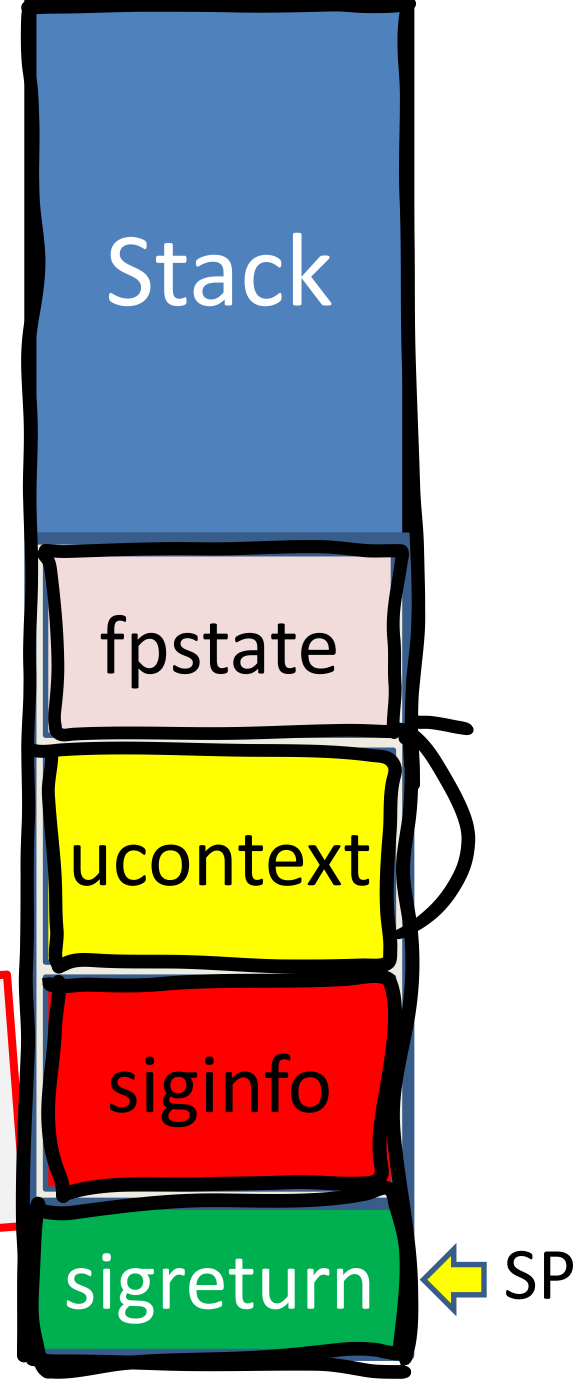


UNIX Signal Handling

- Kernel keeps all administration in userland
  - Good?
  - Bad?

- Kernel does not remember signals

Say attacker fakes signal frame and performs a sigreturn…

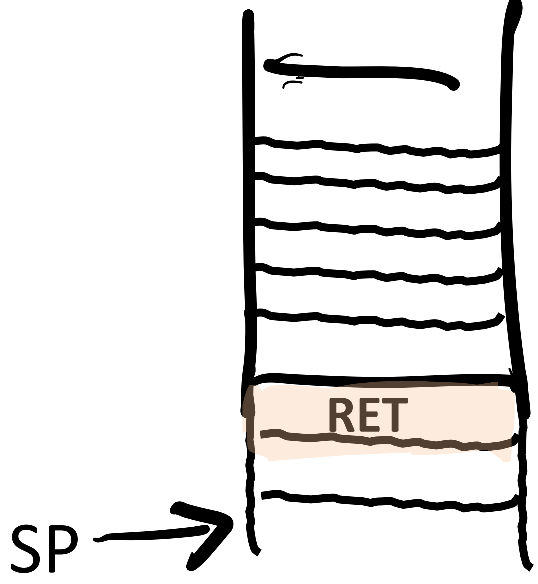


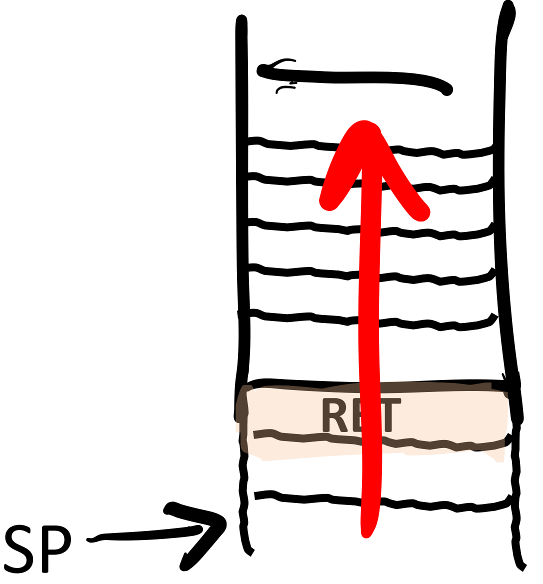
Suppose

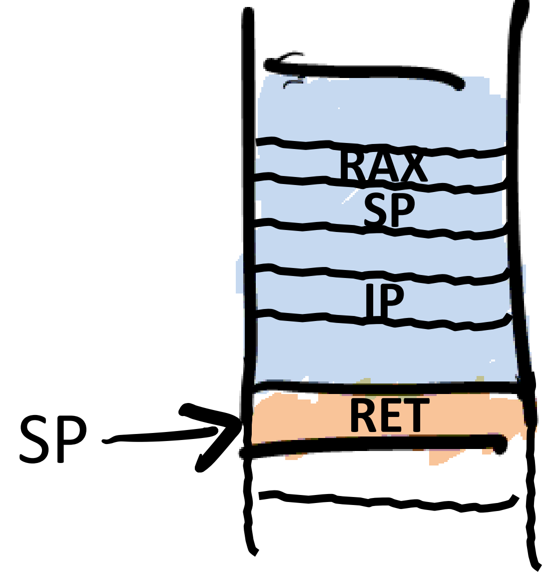
- We have a buffer overflow

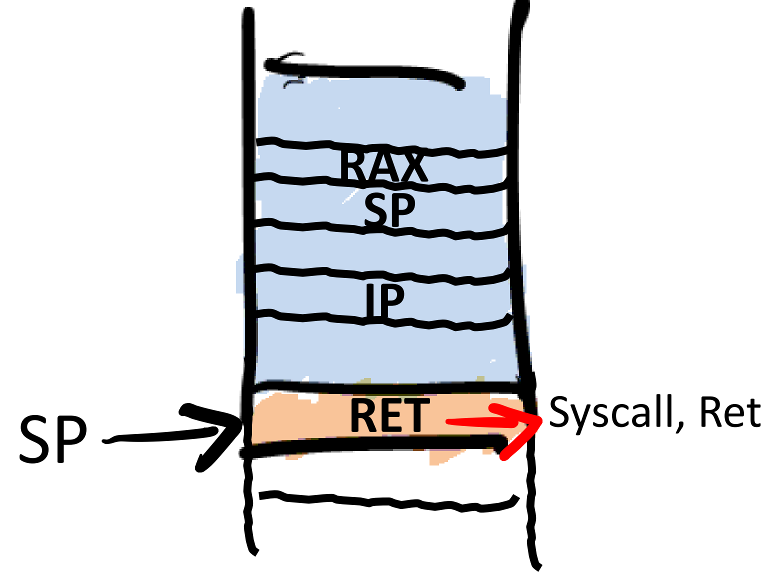
```
void foo (int fd) {
        int len = …;
        char buf[64];
        …
        read (fd, buf+len, 64);
        …
        return;
 }
```

- We can set RAX to 15 (=`sigreturn`)
- We control the stack


RET
SP


RET
SP


RAX
SP
IP
RET
SP


RAX
SP
IP
RET
SP
Syscall, Ret

Generic

- We used SROP for
  - Exploit Asterisk (CVE-2012-5976) on **x86-64Linux**
  - Exploit on **Android**
  - Backdoor on 32- and 64-bit ***Linux**, **BSD**, **Mac OS X**
  - Syscall proxy for **iOS**

And it is Turing Complete!

Generic

- We used SROP for
  - Exploit Asterisk (CVE-2012-5976) on
  - Exploit on

"Weird machine"

And it is Turing Complete!