POKING HOLES IN INFORMATION HIDING



Angelos Oikonomopoulos
Elias Athanasopoulos
Herbert Bos
Cristiano Giuffrida

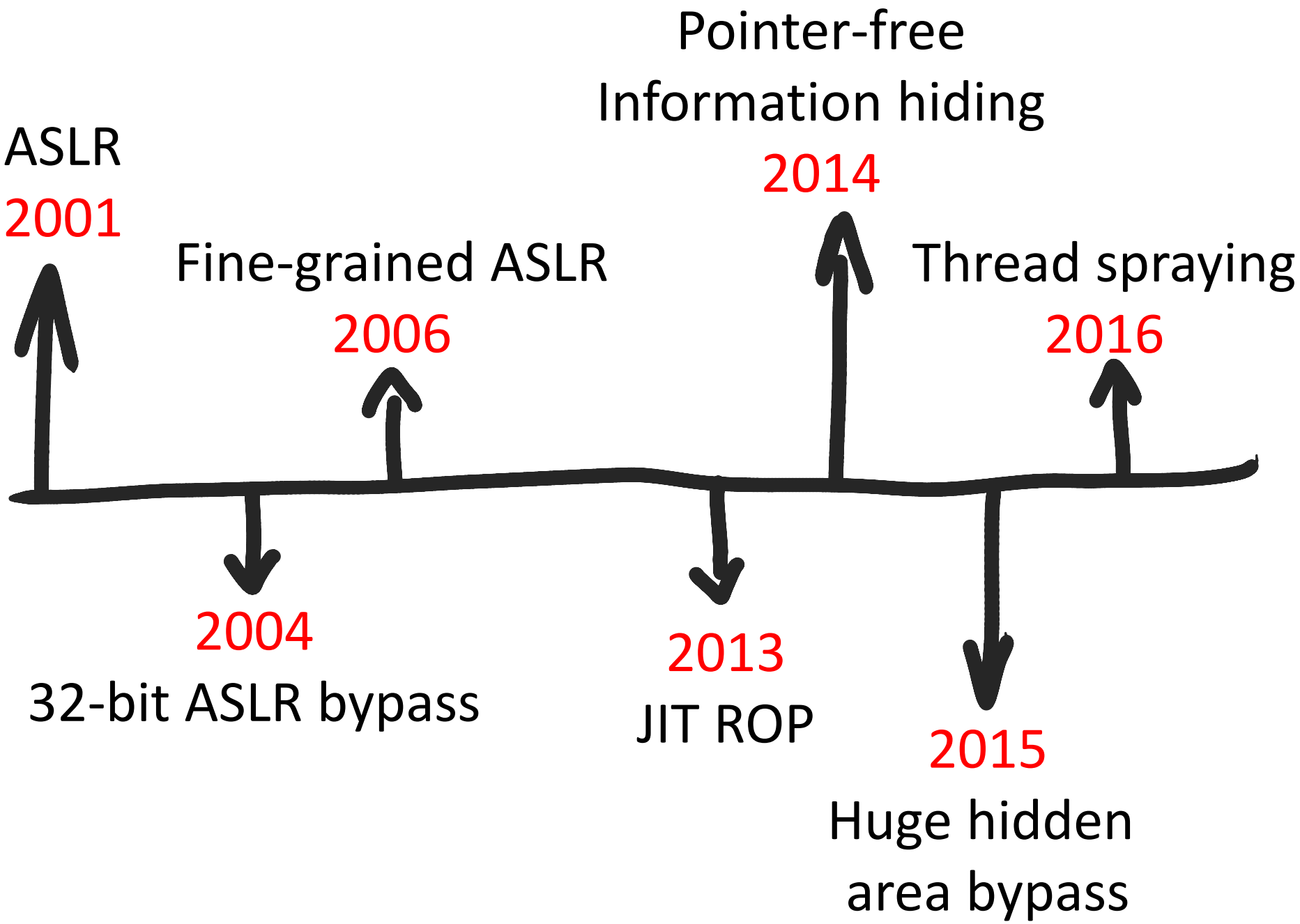
Vrije Universiteit Amsterdam

Teaser

- Break ideal information hiding in seconds
- Few probes, typically no crashes
- Primitives pervasive in server programs

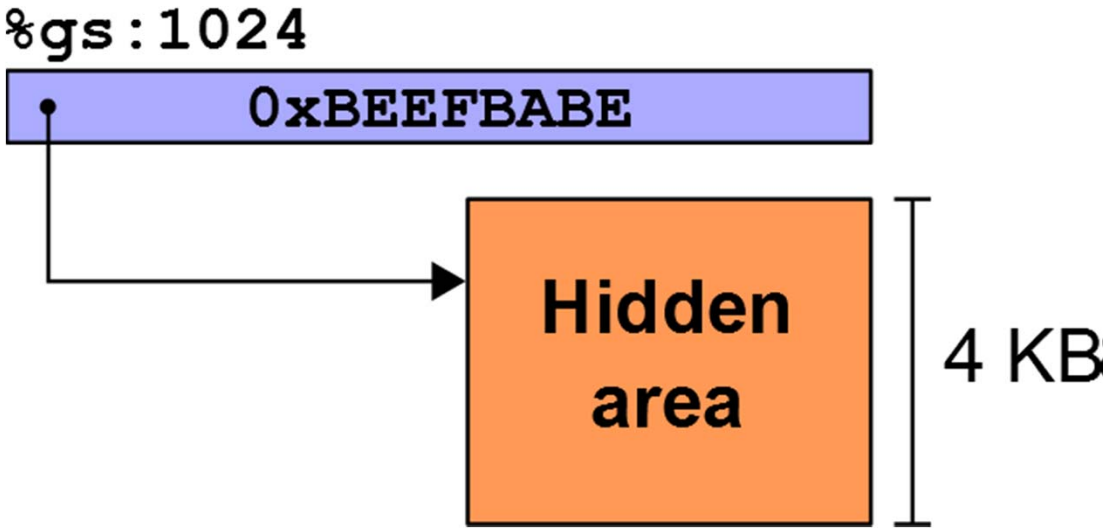
ASLR today

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


ASLR
2001
Fine-grained ASLR
2006
Pointer-free
Information hiding
2014
Thread spraying
2016
2004
32-bit ASLR bypass
2013
JIT ROP
2015
Huge hidden
area bypass

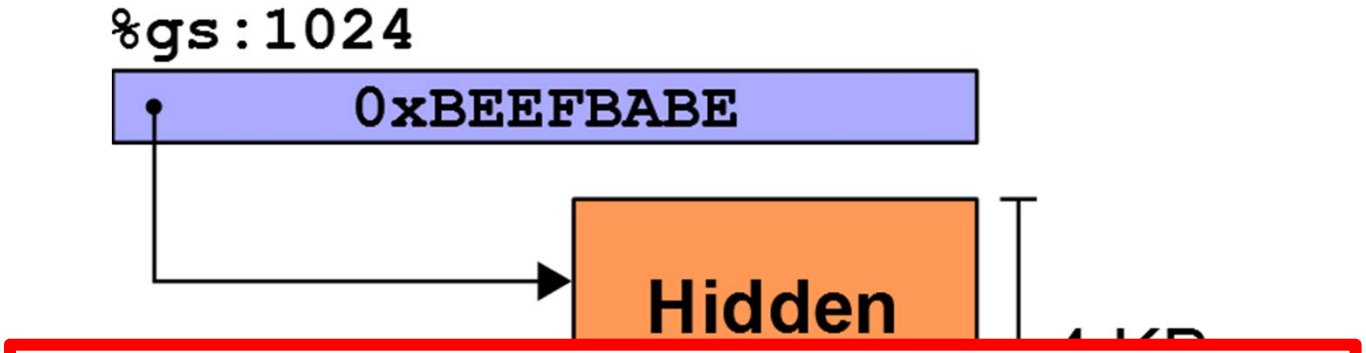
Ideal information hiding

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



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



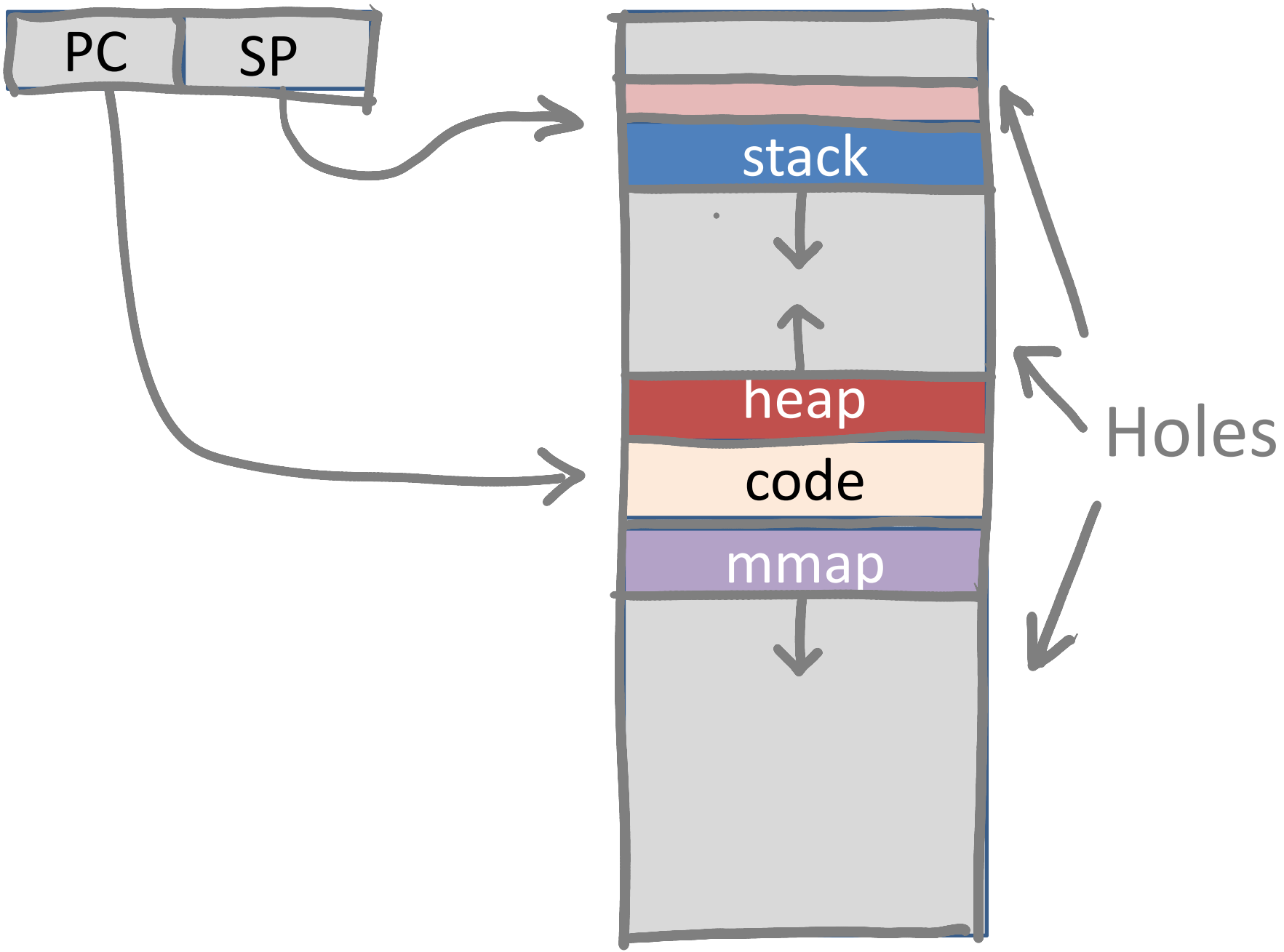
Threat model: arbitrary RW is okay!

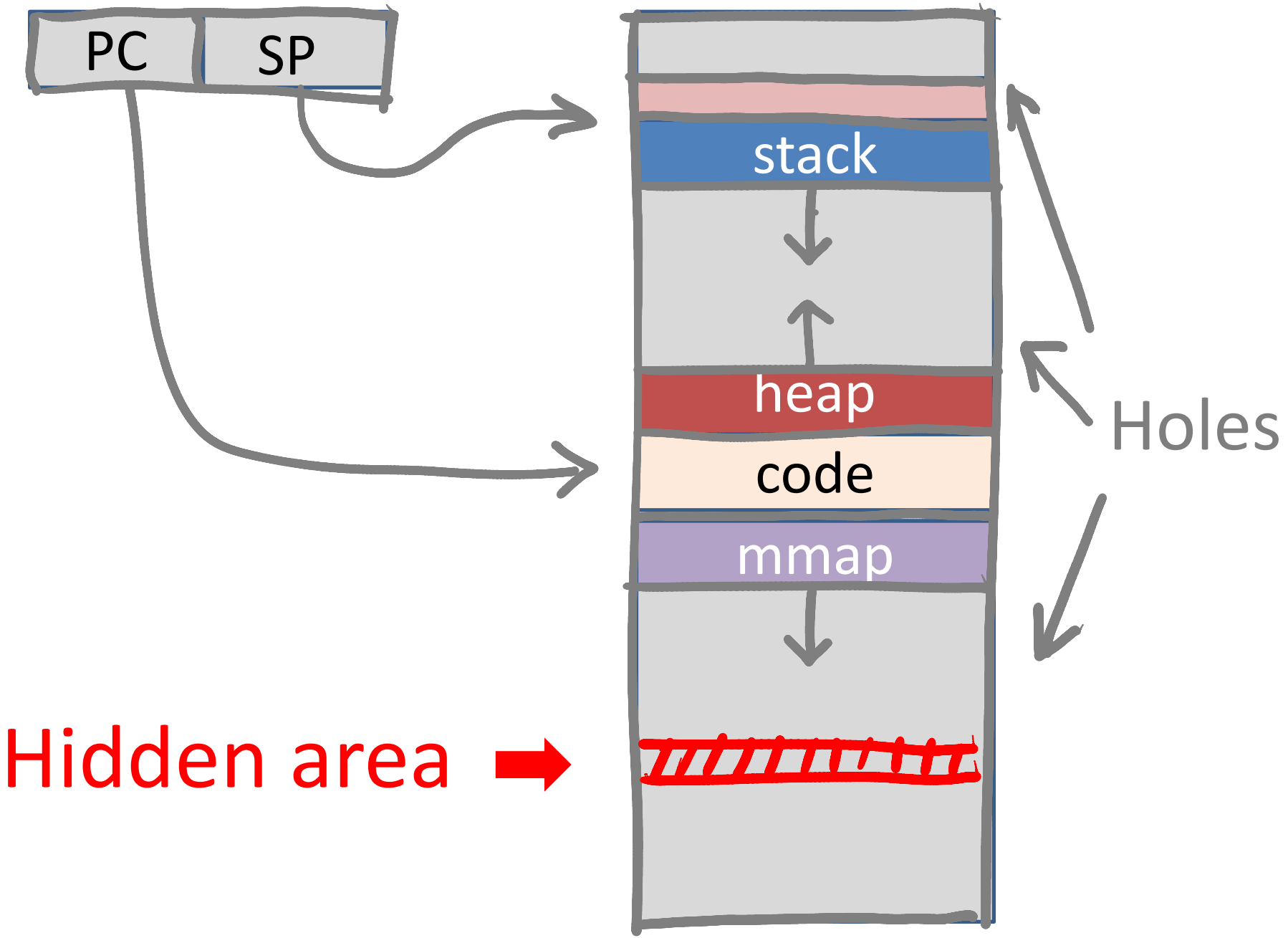
How SAFE?

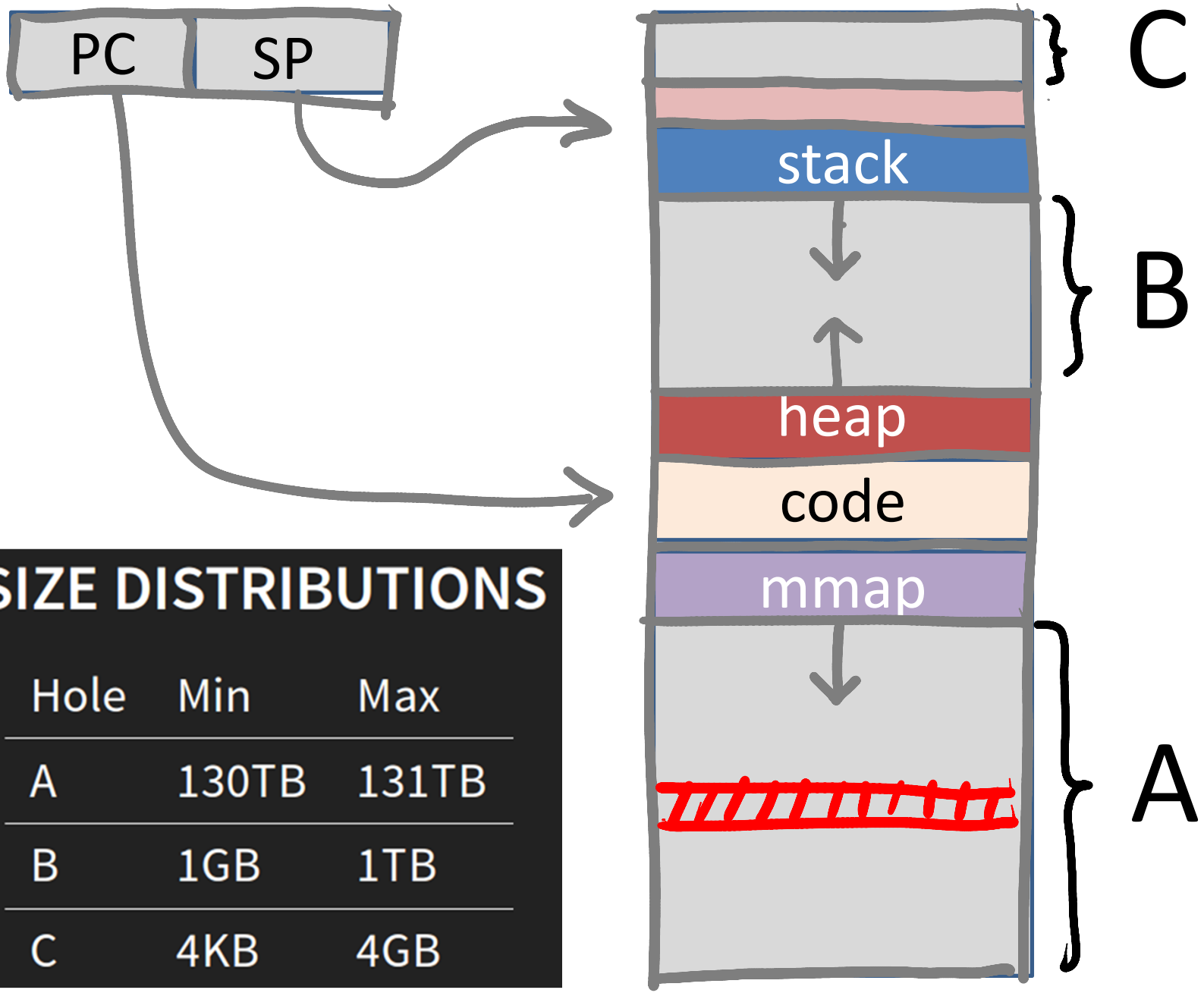
Let’s have a look

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

Let’s not look for the hidden area

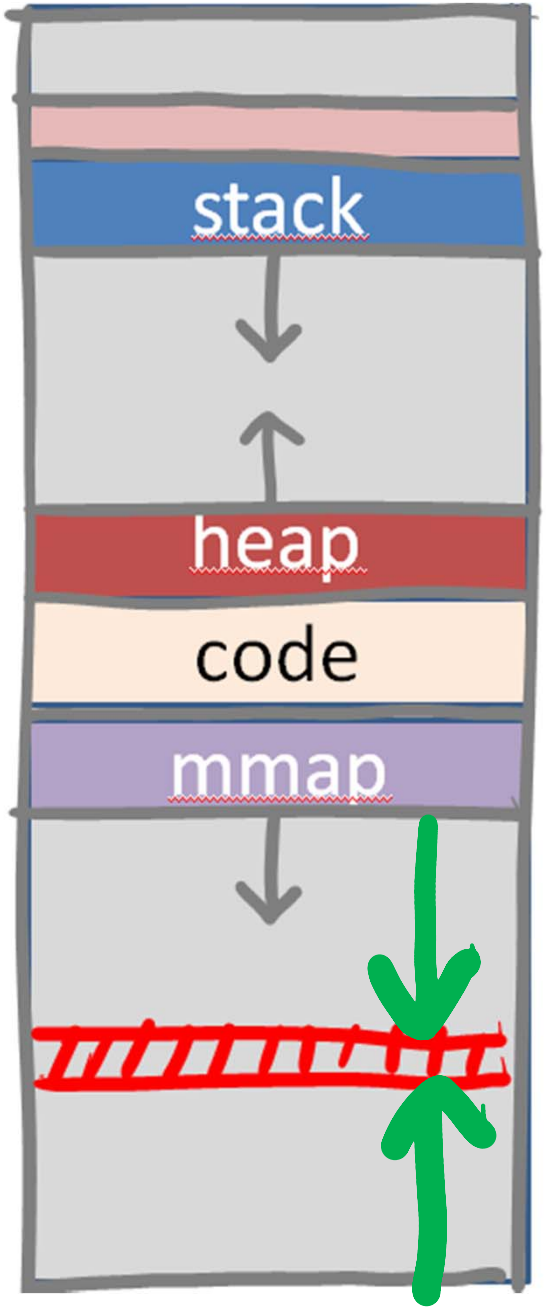
but for the holes!

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!

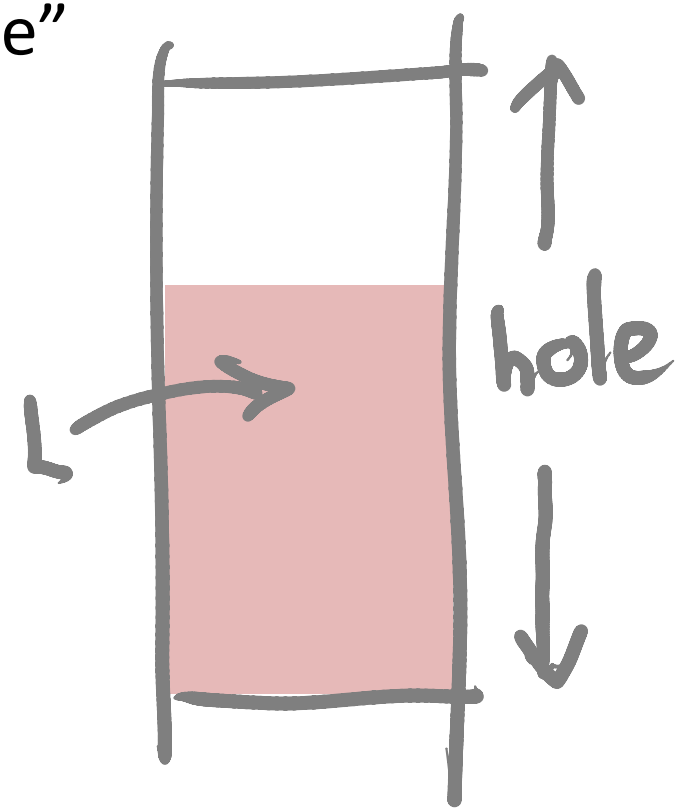


So look for the holes

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

Succeeds!

Sizeof(Hole) ≥ L

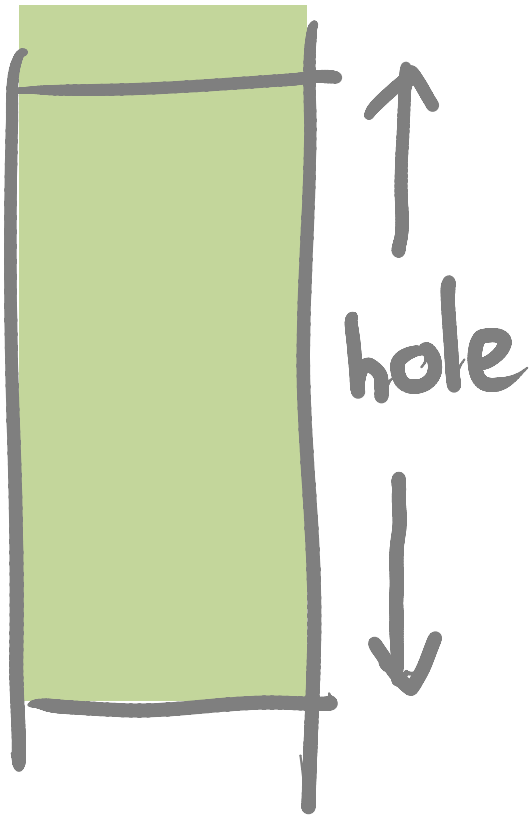


So look for the holes

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

Too large, alloc fails!

Sizeof(Hole) < L

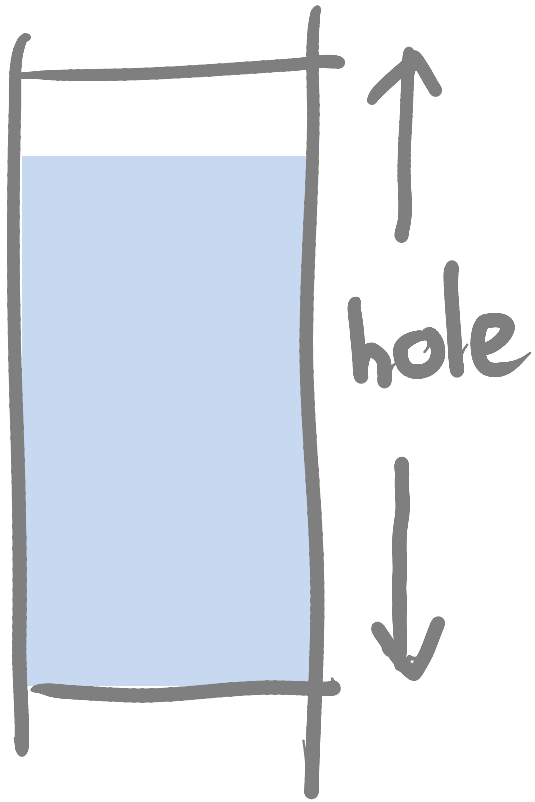


So look for the holes

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

Succeeds!

Sizeof(Hole) ≥ L

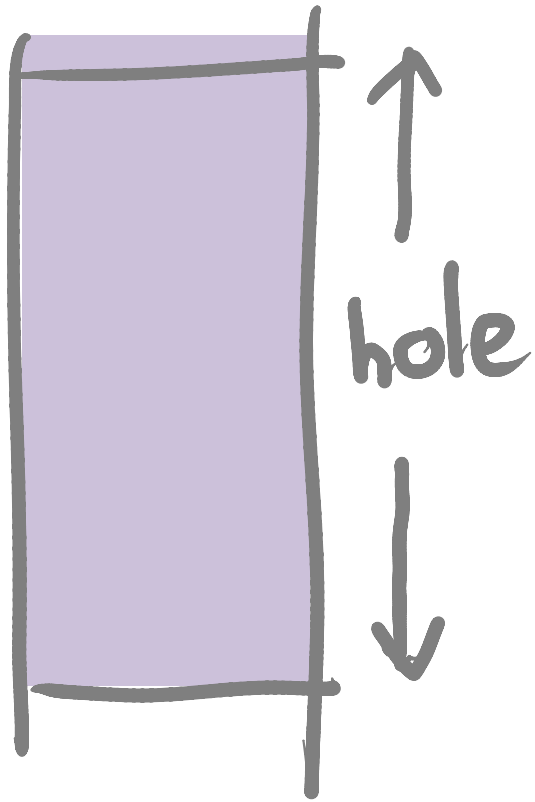


So look for the holes

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

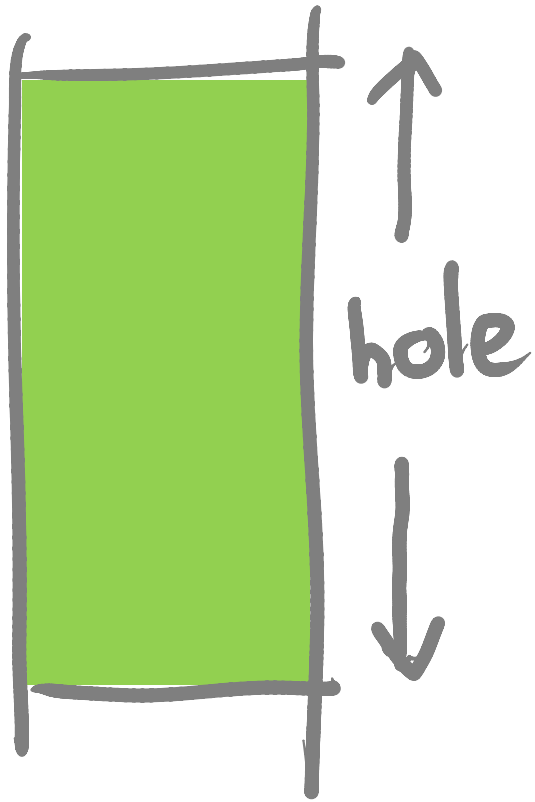
Too large, alloc fails!

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!

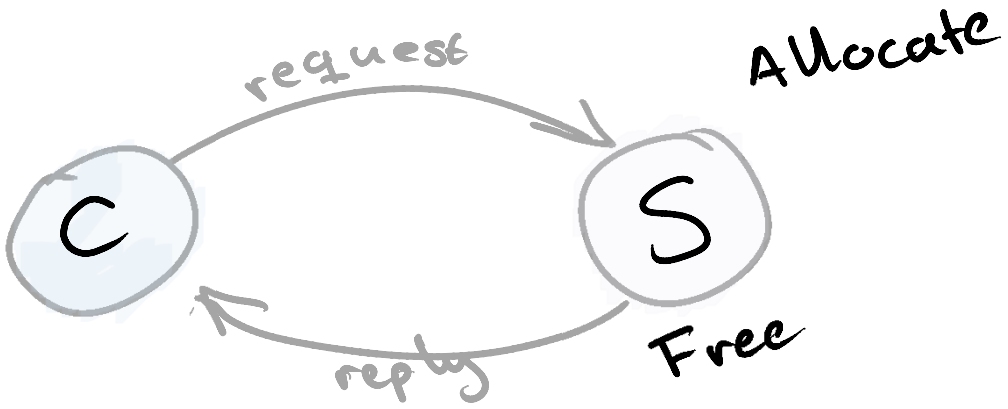
Binary search

Ephemeral Allocation Primitive

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

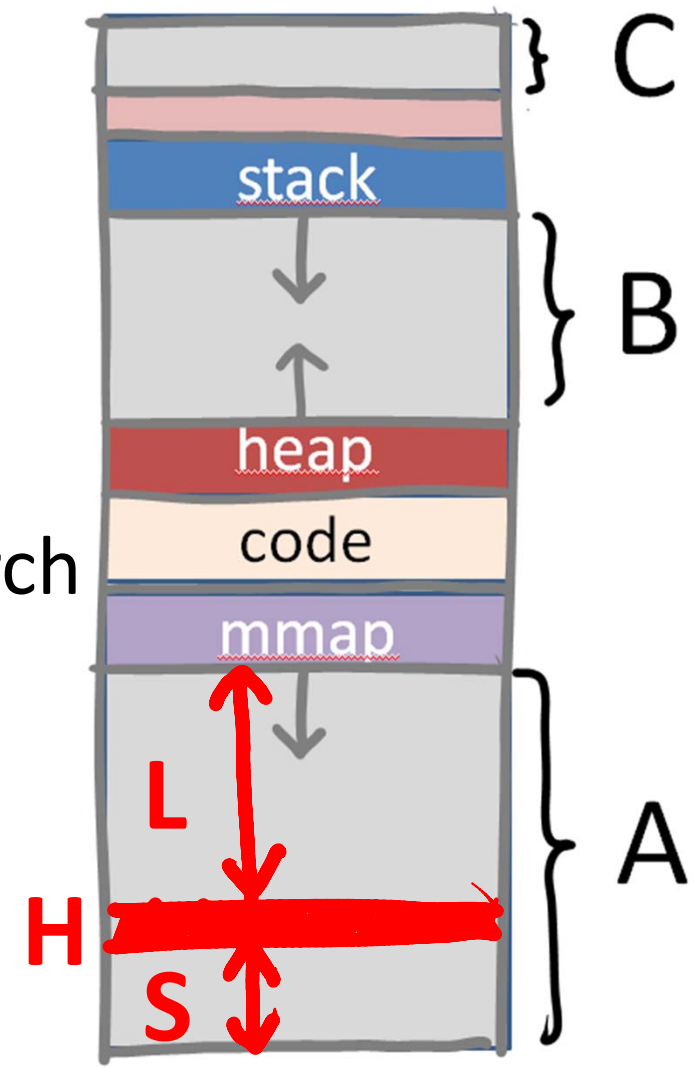
```
ptr = malloc(size);
…
free(ptr);
reply(result);
```

- Strategy: allocation+deallocation, repeat



Ephemeral Allocation Primitive

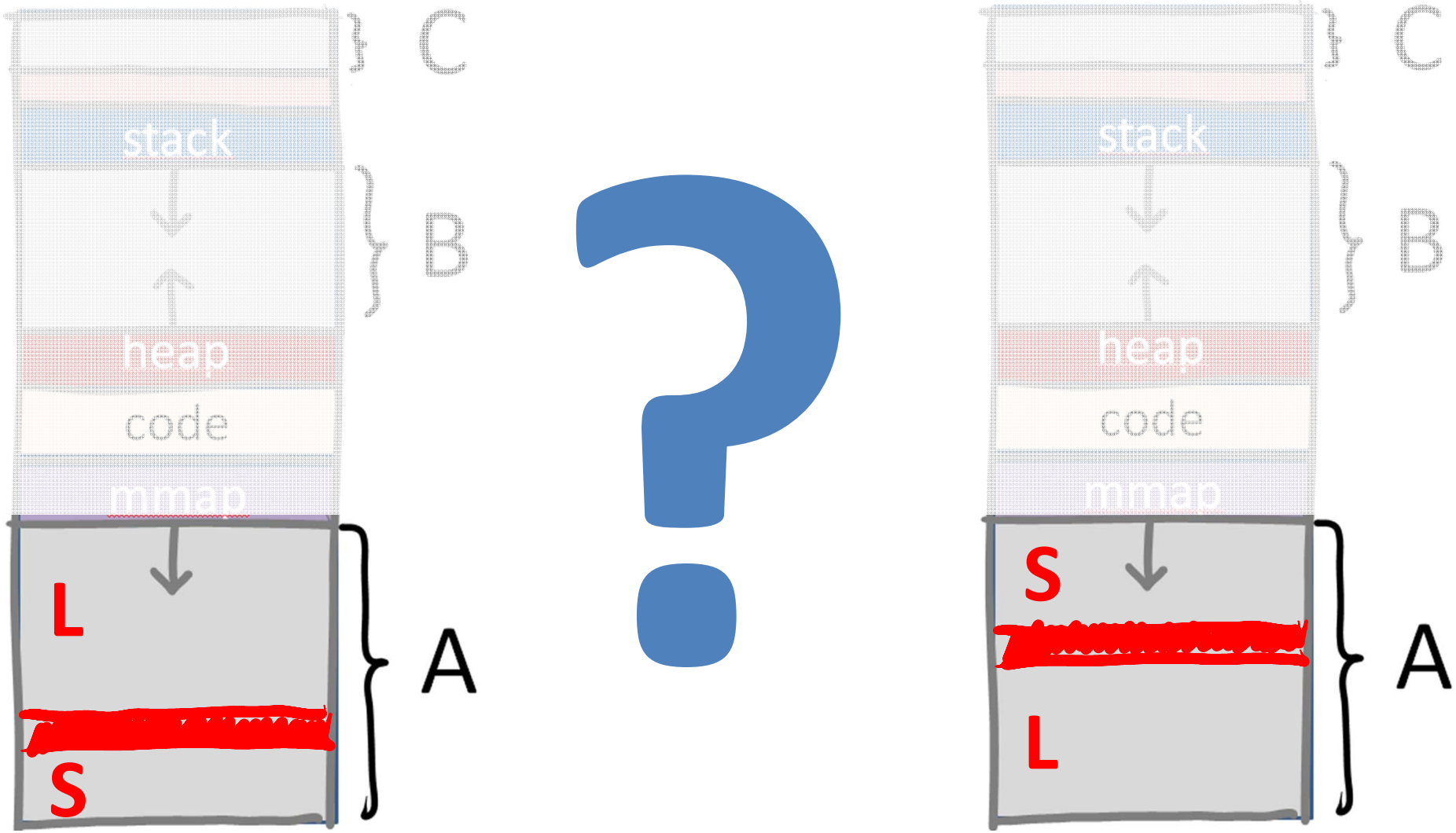
- Say:
  - Single hidden area is in A (*)
  - Hidden area splits A in two
  - L is the largest hole in AS
  - We can find L via binary search



* See paper for generalization

Of course we still miss 1 bit of entropy

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



Would be great

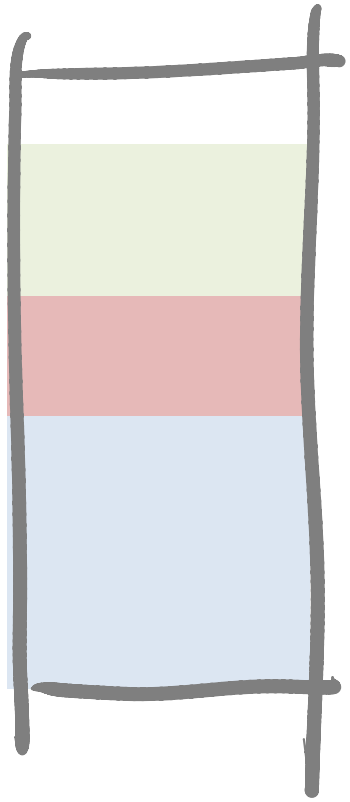
if we could solve this

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

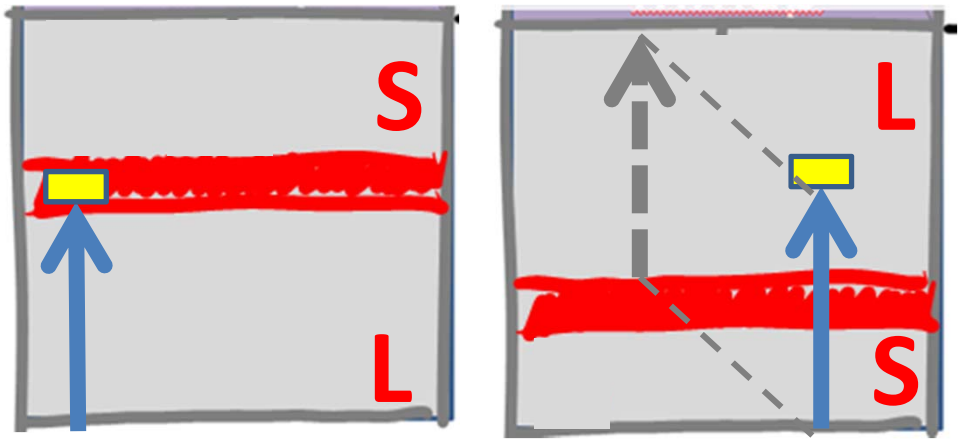


Ephemeral + persistent yields final bit

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:



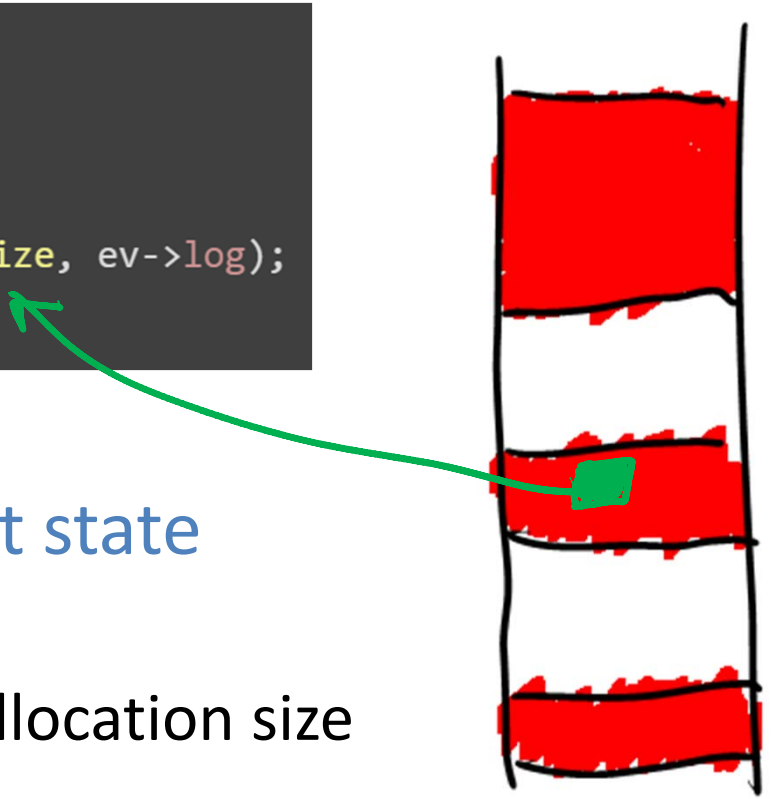
So we need

- 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

Assumption

Memory overcommit:

- OS should allow (virtual) allocations beyond available physical memory
  - Common in server settings
  - Required by some applications:
    - Reddis, Hadoop, virtualization, etc.
- However, even when disabled:
  - Allocation oracles still possible
  - But attacker has to bypass overcommit restrictions

Mitigations

- strict overcommit
  - + reduces attack surface
  - - compatibility issues
- RLIMIT_AS
  - + stops attacks
  - - requires per-application policies
- APM
  - + preserves compatibility
  - - probabilistic

Conclusion

- Allocation oracles, new primitives against ASLR:
  - Efficient
  - Layout-agnostic
  - Pervasive
- Can bypass all information hiding-based defenses
- Even ideal information hiding is insufficient
- Time for better (meta)data protection techniques
  - More info: https://www.vusec.net/nowhere-to-hide

