CS255: Computer Security

Memory Safety

Chengyu Song 01/24/2022

Memory Errors

- Spatial errors: out-of-bound memory access
  - Stack buffer overflow
  - HeartBleed
- Temporal erros
  - Use-before-initialization (UBI)
  - Use-after-free (UAF)

HeartBleed

A simple bug in the OpenSSL library

- A out-of-bound memory read vulnerability in the implementation of the heartbeat extension (RFC6520) of the TLS (Transportation Layer Security) protocol
- Allows attackers to steal sensitive information from the vulnerable website (e.g., the private key of a X509 certificate)
- It was introduced into the software in 2012 and publicly disclosed in April 2014

HeartBleed

Impacts

System administrators were frequently slow to patch their systems. As of 20 May 2014, 1.5% of the 800,000 most popular TLS-enabled websites were still vulnerable to Heartbleed.[9] As of 21 June 2014, 309,197 public web servers remained vulnerable.[10] As of 23 January 2017, according to a report[11] from Shodan, nearly 180,000 internet-connected devices were still vulnerable.[12][13] As of 6 July 2017, the number had dropped to 144,000, according to a search on shodan.io for "vuln:cve-2014-0160".[14] As of 11 July 2019, Shodan reported[15] that 91,063 devices were vulnerable. The U.S. was first with 21,258 (23%), the top 10 countries had 56,537 (62%), and the remaining countries had 34,526 (38%). The report also broke the devices down by 10 other categories such as organization (the top 3 were wireless companies), product (Apache httpd, nginx), or service (https, 81%).

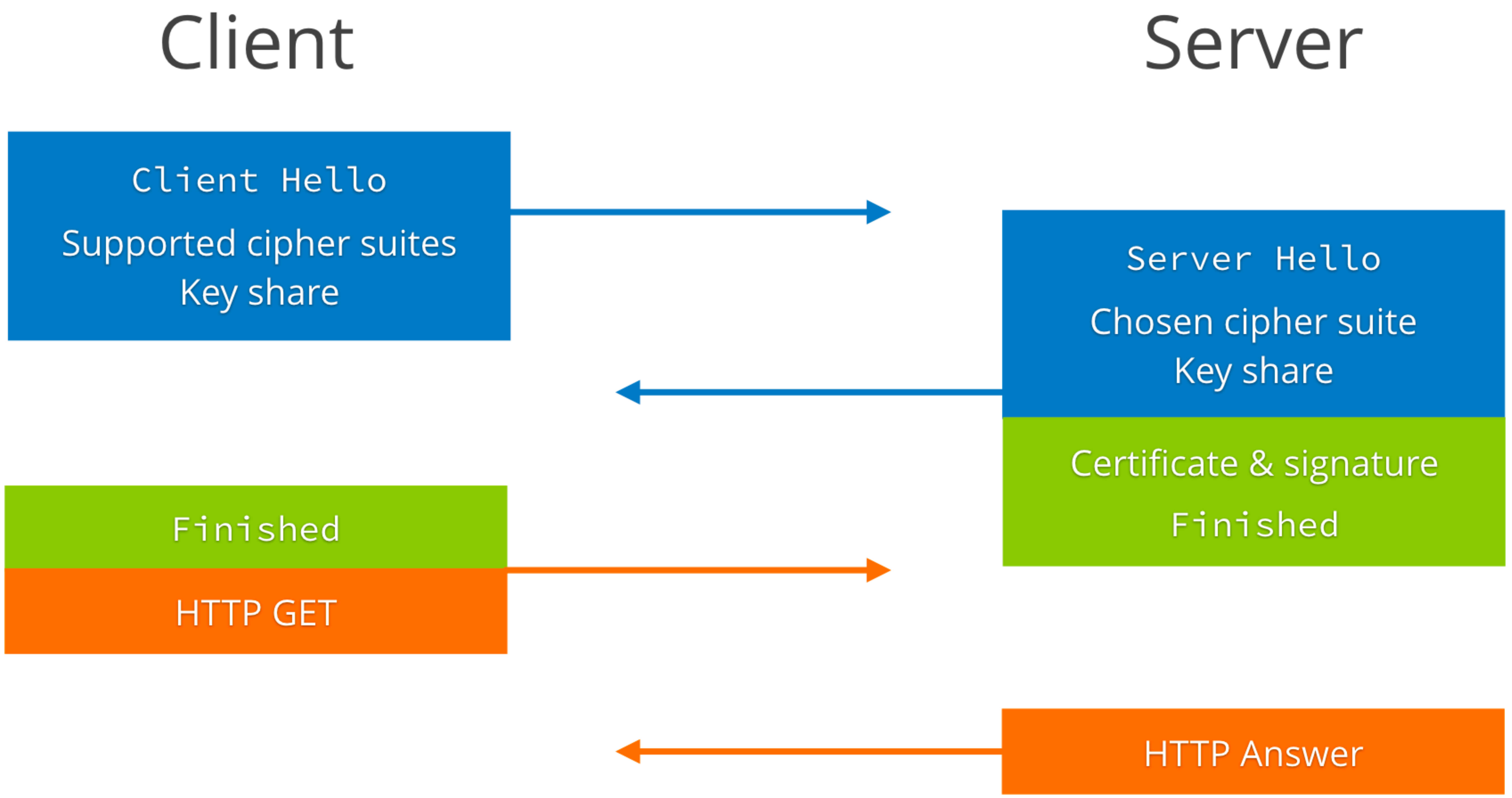
HeartBleed

Background

- Transportation Layer Security (TLS) protocol (RFC 8446)
  - A cryptographic protocol for secure communication
  - Two sub-protocols
    - Handshake Protocol: for authentication
    - Record Protocol: for confidentiality and integrity
  - The underlying protocol of https://

HeartBleed

The TLS Handshake Protocol

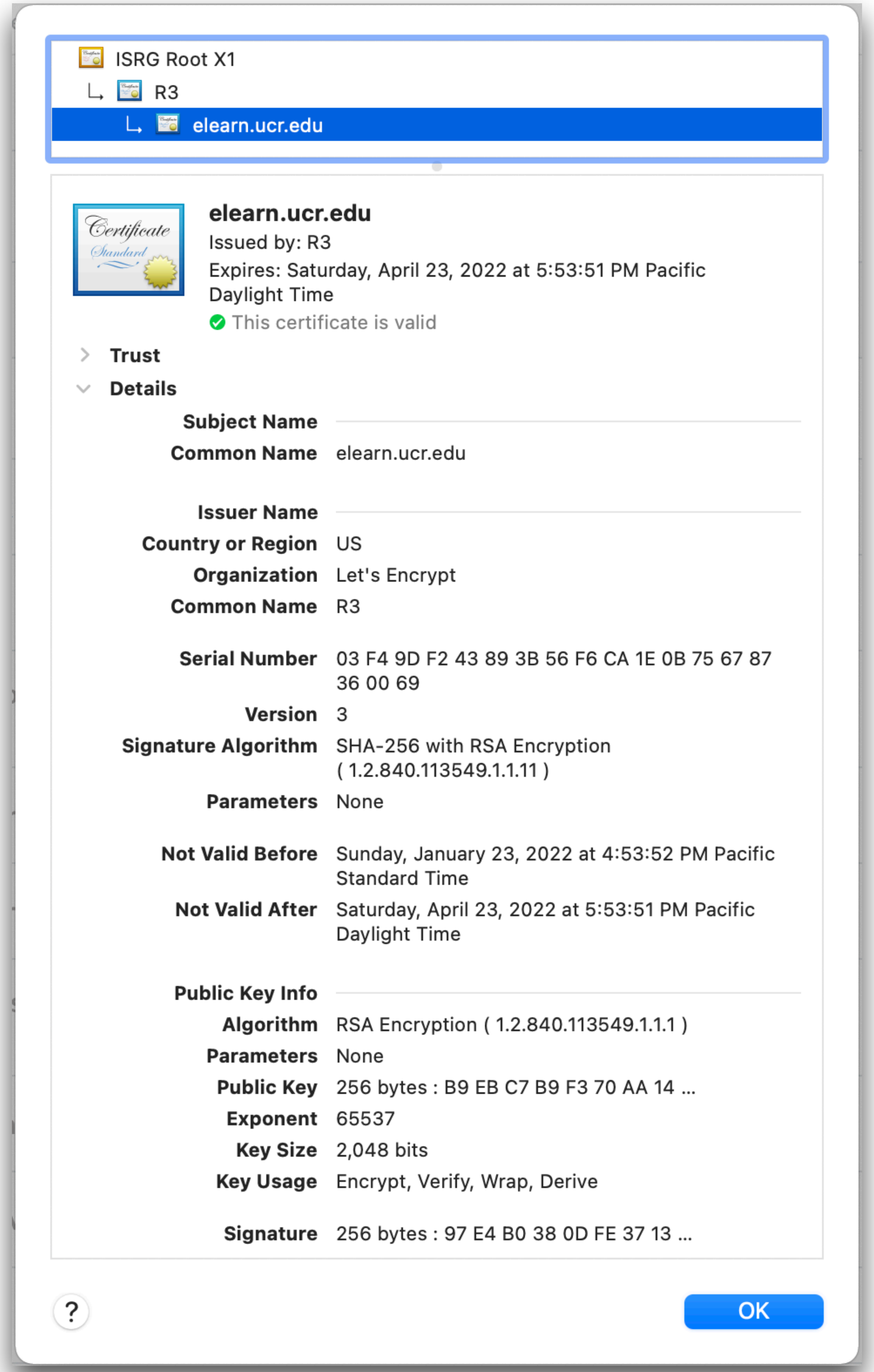


- Verify the identify of the server [and the client]
- Exchange a secret to derive the session key for the Record Protocol

HeartBleed

How authentication is done

- Based on public key cryptographic



HeartBleed

The TLS Record Protocol

TLS record format, general

Offset	Byte +0	Byte +1	Byte +2	Byte +3
Byte 0	Content type	N/A		
Bytes 1..4	Legacy version		Length	
	(Major)	*(Minor)*	*(bits 15..8)*	*(bits 7..0)*
Bytes 5..(m–1)	Protocol message(s)			
Bytes m..(p–1)	MAC (optional)			
Bytes p..(q–1)	Padding (block ciphers only)			

HeartBleed

The HeartBeat Extension

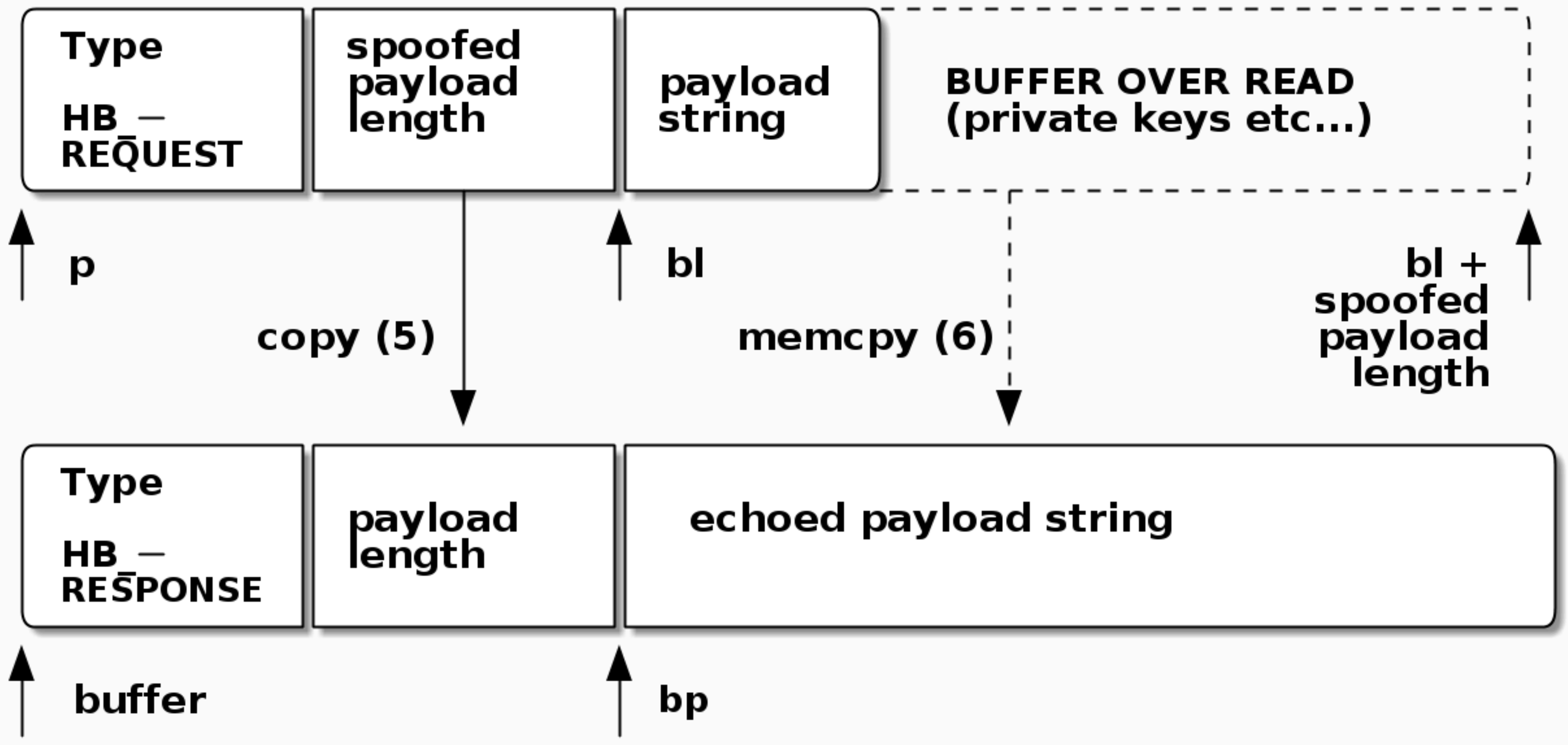
- Motivation: how to know if the peer is still alive
    - Renegotiation (handshake) is expensive
- Solution: a heartbeat message
    - The Heartbeat protocol messages consist of their type and an **arbitrary** payload and random padding of at least 16 bytes
    - When a HeartbeatRequest message is received and sending a HeartbeatResponse is not prohibited as described elsewhere in this document, the receiver MUST send a corresponding HeartbeatResponse message carrying an exact copy of the payload of the received HeartbeatRequest

HeartBleed

The vulnerability

```
struct {
   HeartbeatMessageType type;
   uint16 payload_length;
   opaque payload[HeartbeatMessage.payload_length];
   opaque padding[padding_length];
} HeartbeatMessage;
```

- Could you image what is the bug/ vulnerability?



Spatial Memory Errors

Definition

- Spatial Memory Errors occur when the access is out-of-bound
- How to define the bound?
  - A1: pointer as a capability —> SoftBound
  - A2: undefined memory —> AddressSanitizer

Pointer as a Capability

Creation of pointers

- What are legitimate ways to create pointers?
  - Allocation
    - Stack and global: declaration means allocation
    - Heap: explicit (e.g., malloc)
  - Address taken
    - of code: fp = &func
    - of data: p = &d

Pointer as a Capability

Creation of pointers

- Propagation
    - p1 = p2
- Pointer arithmetic
    - p = &array[index]
    - p = &struct->field
- Type casting
    - p1 = type_cast(p2)

Pointer as a Capability

How to track capabilities

- Fat pointer: p := {bounds, address}
  - Fastest bounds lookup, but breaks binary compatibility
- Lotfat pointer: p := {meta_addr, address}
  - Faster bounds lookup, but requires special memory layout
- Decoupled metadata: meta(p) = lookup(p)
  - Slow bounds lookup, but has good binary compatibility

Pointer as a Capability

Capability reduction

- What is the expected capability of a pointer?
  - Based on allocation size?
  - Based on type?
- A combination of both: whichever is smaller

Pointer as a Capability

Challenges

- Type casting: how to recover (allocation) capabilities
  - Track the allocation type (e.g., EffectiveSan)
- Different capabilities for different operations
  - char *p = “abc”; *p; p++;
- Atomicity
  - How to make sure (decoupled) capabilities are always sync with the pointer

Pointer as a Capability

Capability forgery

- Recall our stack buffer overflow case, what did we forge?

```
bottom of                                                      top of
memory                                                         memory
                   buffer            sfp   ret   *str
<=-----           [AAAAAAAAAAAAAAAA][AAAA][AAAA][AAAA]

top of                                                      bottom of
stack                                                           stack
```

Pointer as a Capability

How to prevent forgery?

- Encryption: PointerGuard, Pointer Authentication Code (PAC)
  - Usually not strong enough
- Tagged memory: the CHERI architecture
  - Requires hardware changes
- Decoupled and protected metadata: SoftBound, Intel Memory Protection Extension (MPX)

Pointer as a Capability

Capability Revocation

- When a memory object is freed, all pointers point to the region should become invalid
- **Dangling pointers**: pointers point to freed memory objects (the whole region)
- UAF: deference a dangling pointer
  - Dangling pointers are common, but UAF is much rare
  - How to exploit a UAF vulnerability?

Pointer as a Capability

Capability revocation

- Nullification: p = NULL
  - Automated pointer nullification
- Key/version invalidation: key(p) != key(m)
  - Each pointer and memory has a key/version (e.g., using memory tags)
- Delayed free
  - Conservative garbage collection

Accessing Undefined Memory

Address Sanitizer

- Undefined memory (redzones) is not allowed to access
- What regions are undefined?
    - Spatial: out-of-bound regions —> insert redzones between allocated memory objects
    - Temporal: freed regions mark freed objects as redzones

Accessing Undefined Memory

Address Sanitizer: shadow memory

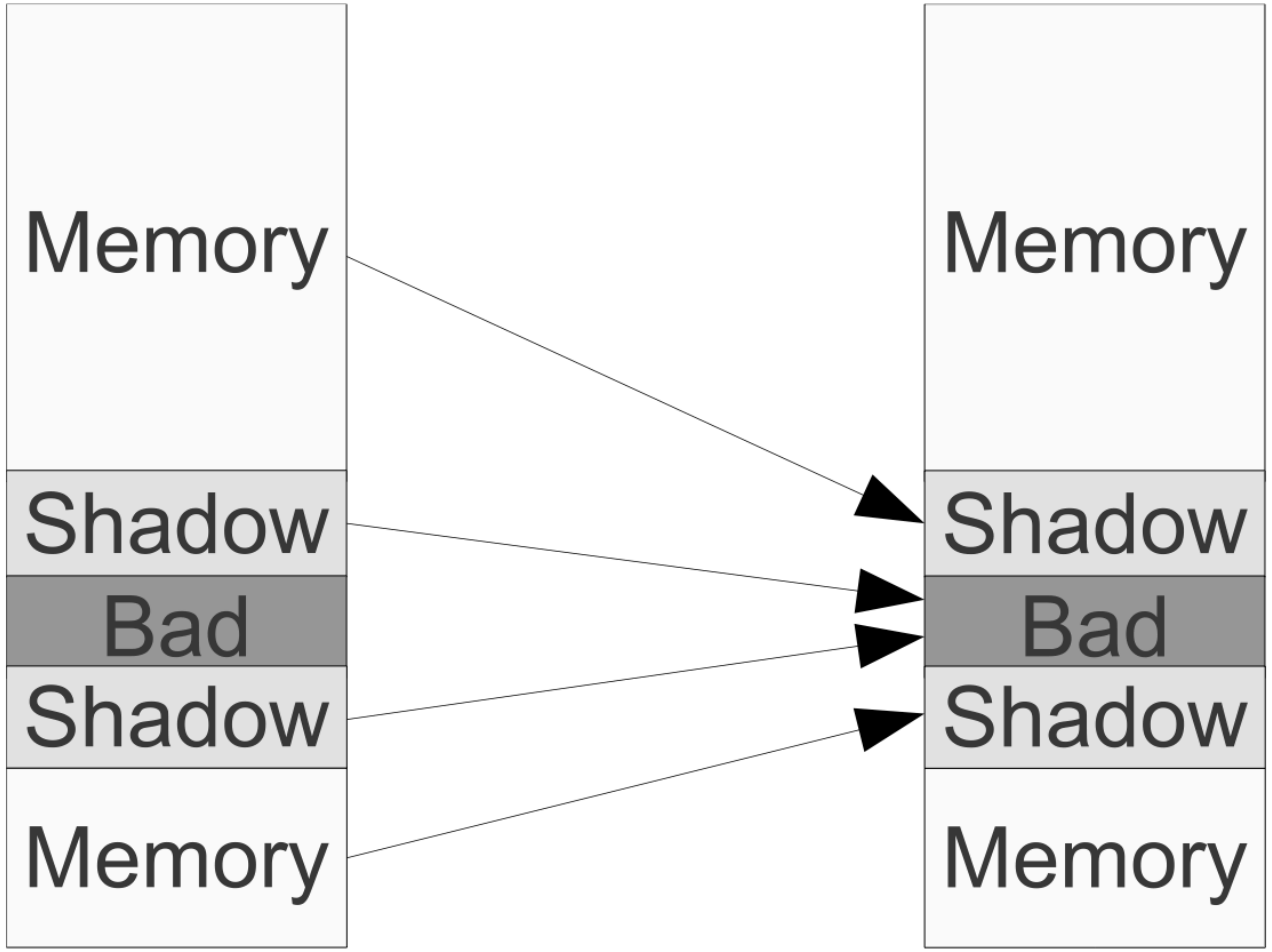


Figure 1: AddressSanitizer memory mapping.

Accessing Undefined Memory

Address Sanitizer

- Advantages
  - Compatibility: user-mode programs, kernel, even binaries
- Bypassable
  - Spatial safety demands infinite "gap" (redzone) between memory objects
  - Temporal safety demands freed regions should never be reused

Use-Before-Initialization

- Uninitialized pointer
  - Simple: no associated capability, dereference is invalid
- Uninitialized data
  - Hard: similar to dangling pointers
- How to exploit UBI vulnerabilities?
- How to mitigate UBI vulnerabilities?
  - Forced initialization

Why Memory Safety

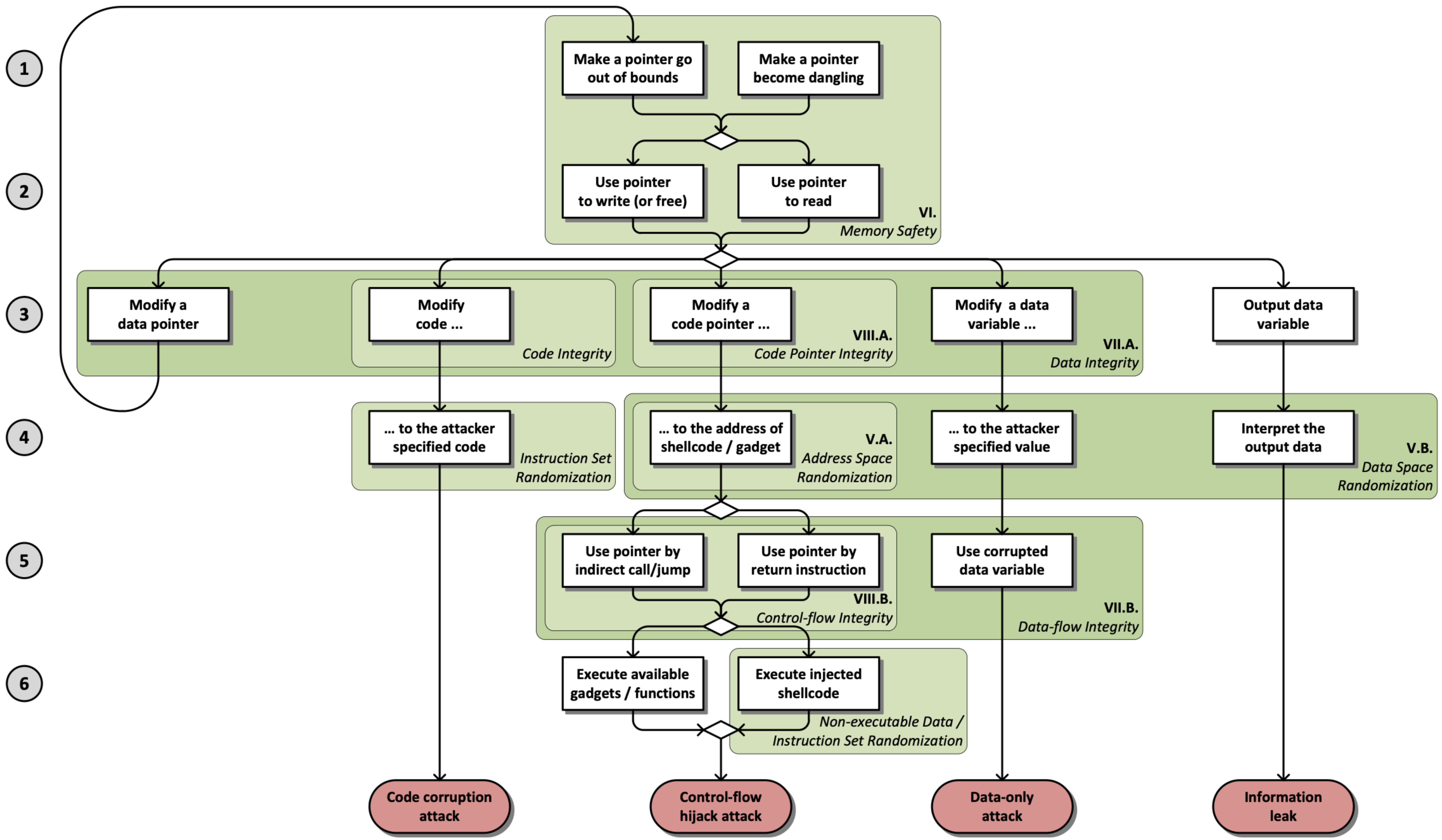


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

Why NOT Memory Safety?

- Compatibility: C/C++ is too flexible so retrofitting memory safety into legacy code is likely to create compatibility problem
  - SoftBound can only compile a small subset of SPEC CPU benchmarks
  - Intel MPX is being abandoned by GCC and Linux
- Performance overhead
  - Metadata lookup
  - Capability checks

Best Option so far

- Use a memory safe program language
    - Rust
    - Go
    - Java