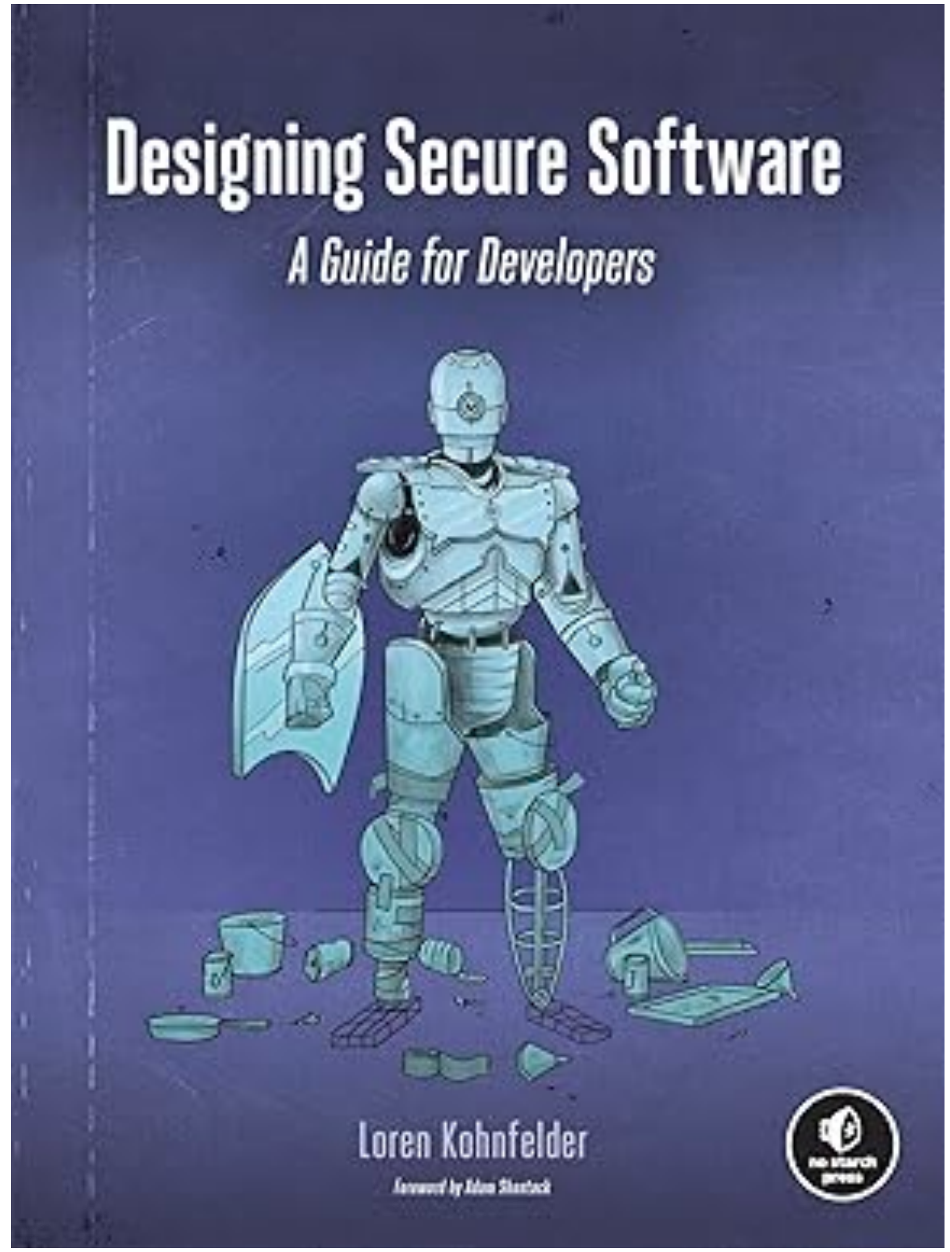


9 Low-Level Coding Flaws

Topics

- Arithmetic Vulnerabilities
  - Fixed-Width Integer Vulnerabilities
  - Floating-Point Precision Vulnerabilities
  - Example: Floating-Point Underflow
  - Example: Integer Overflow
  - Safe Arithmetic

Topics

- Memory Access Vulnerabilities
  - Memory Management
  - Buffer Overflow
  - Example: Memory Allocation Vulnerabilities
  - Case Study: Heartbleed

Arithmetic Vulnerabilities

Fixed-Width Integer Vulnerabilities

- 16-bit integer
- Possible values from
  - 0000 0000 0000 0000 = 0
  - to
  - 1111 1111 1111 1111 = 65,535 = 2^{16} - 1
- Multiply 300 x 300 = 90,000
  - 24,464 in a 16-bit integer
  - 65, 536 + 24,464

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

$$0000000100101100 = 300$$

$$(2^8 + 2^5 + 2^3 + 2^2) = 300$$

Now let's see how to multiply 300 times itself in binary (*Figure 9-2*).

```
         15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
         0000000100101100
       x 0000000100101100
         ----------------
      00 00000100101100
     000 0000100101100
   00000 00100101100
00000001 00101100
         ----------------
         0101111110010000
```

Consequences of Integer Overflows

- Buffer overflow
- Incorrect comparison of values
- Giving a credit instead of charging for a sale
- etc.

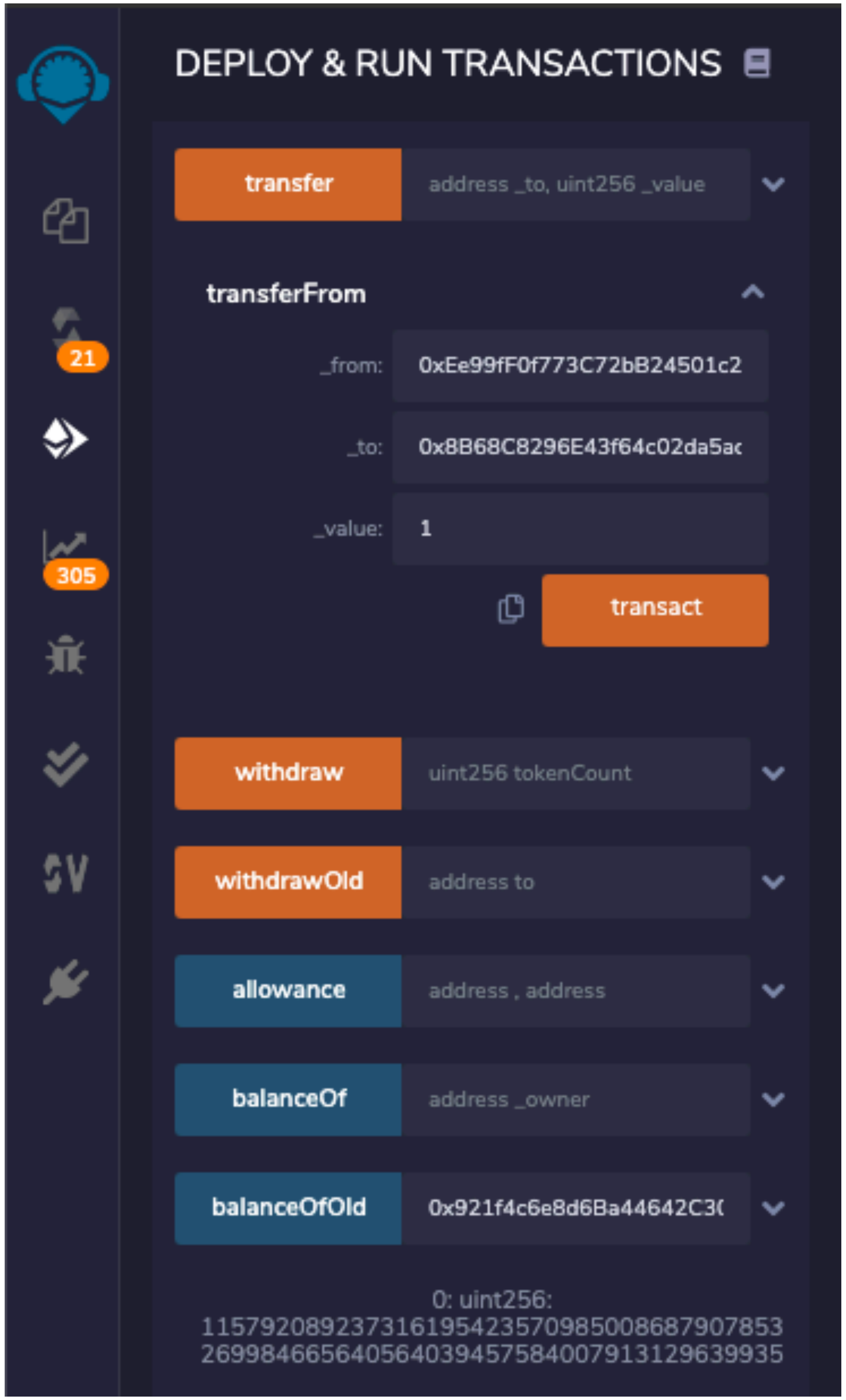
How $800k Evaporated from the PoWH Coin Ponzi Scheme Overnight

Eric Banisadr · Follow
4 min read · Feb 1, 2018

- A flaw in the smart contract allowed an attacker to subtract 1 coin from a wallet containing zero coins

- The resulting balance: $2^{256} - 1$

```
0: uint256:
115792089237316195423570985008687907853
269984665640564039457584007913129639935
```


DEPLOY & RUN TRANSACTIONS
21
305
transfer
address _to, uint256 _value
transferFrom
_from:
0xEe99fF0f773C72bB24501c2
_to:
0x8B68C8296E43f64c02da5ac
_value:
1
transact
withdraw
uint256 tokenCount
withdrawOld
address to
allowance
address , address
balanceOf
address _owner
balanceOfOld
0x921f4c6e8d6Ba44642C3(
0: uint256:
115792089237316195423570985008687907853
269984665640564039457584007913129639935

Avoiding Integer Overflows and Underflows

- Use an integer size larger than the largest allowable value
  - Preceded by checks ensuring that invalid values never sneak in
- When multiplying two 16-bit numbers, put the result in a 32-bit number
- Or use memory-safe languages like Rust

```
cnit_123a@deb:~/Ov-App$ ./Ov
x is 231
x is 233
x is 236
x is 240
x is 245
x is 251
thread 'main' panicked at 'attempt to add with overflow'
```

Floating-Point Precision Vulnerabilities

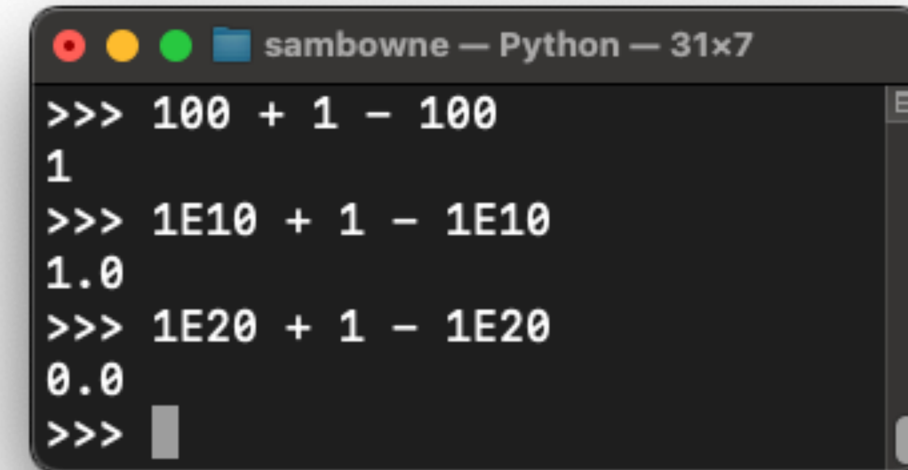
- Numbers like 1.543E23
  - 1,543 x 10^{23}
- Three parts
  - A sign bit (+ or -)
  - A fraction (15 digits in precision)
  - An exponent

Tiny Errors

- 0.1 + 0.2 will yield 0.30000000000000004
- **Solutions**
  - Use integer arithmetic (good for money, count pennies)
  - Don't use **(x == y)** for floating points
  - Use **(x > y - delta && x < y + delta)**
  - Or the Python function **math.isclose()**

Example: Floating-Point Underflow

- Floating point numbers only have a limited number of decimal places of precision
- The 1 is lost, below that precision level
- **Solution**
  - Limit the values to a maximum that's safe
  - Such as 1E10



Example: Integer Overflow

- To calculate hourly pay in C
  - Use 32-bit integers (maximum 4 billion)
- Code time as millihours (8000 = 8 hours)
- Dollar values in cents ($400 = 40,000 cents, the maximum possible pay)
- A week's work would be 40 hours
  - 40,000 millihours x 40,000 cents = 1,600,000,000
  - Very close to the limit
  - Adding overtime pay can easily exceed 32-bit limit
- This task requires 64-bit integers

Safe Arithmetic

- Avoid using tricky code to handle overflow problems
  - Mistakes will be hard to find
  - And tricks may depend on the implementation on your machine

Safe Arithmetic

- Be careful using type conversions that can potentially truncate or distort results, just as calculations can.
- Where possible, constrain inputs to the computation to ensure that all possible values are representable.
- Use a larger fixed-size integer to avoid possible overflow; check that the result is within bounds before converting it back to a smaller-sized integer.
- Remember that intermediate computed values may overflow, causing a problem, even if the final result is always within range.
- Use extra care when checking the correctness of arithmetic in and around security-sensitive code.

- https://www.destroyallsoftware.com/talks/wat

Memory Access Vulnerabilities

Memory Management

- Pointers allow direct access to memory by address
    - A powerful, but dangerous, feature of C
- **alloc()** reserves memory on the heap
- **free()** frees it

Correct Heap Usage

```
uint8_t *p;
// Don't use the pointer
// before allocating memory for it.

p = malloc(100);  // Allocate 100 bytes
                  // before first use.
p[0] = 1;
p[99] = 123 + p[0];

free(p);          // Release the memory
                  // after last use.

// Don't use the pointer anymore.
```

Vulnerabilities

- Dangling pointer
  - Can read or write to pointer after it's freed, or no longer intended to be used
- Double-free
  - Second free operation causes improper, dangerous write operations

Buffer Overflow

```
char password[10];      // Reserve 10 bytes

scanf("%s", password); // Can read more
                       // than 10 bytes
```

- Can write outside reserved memory
- Allows code injection

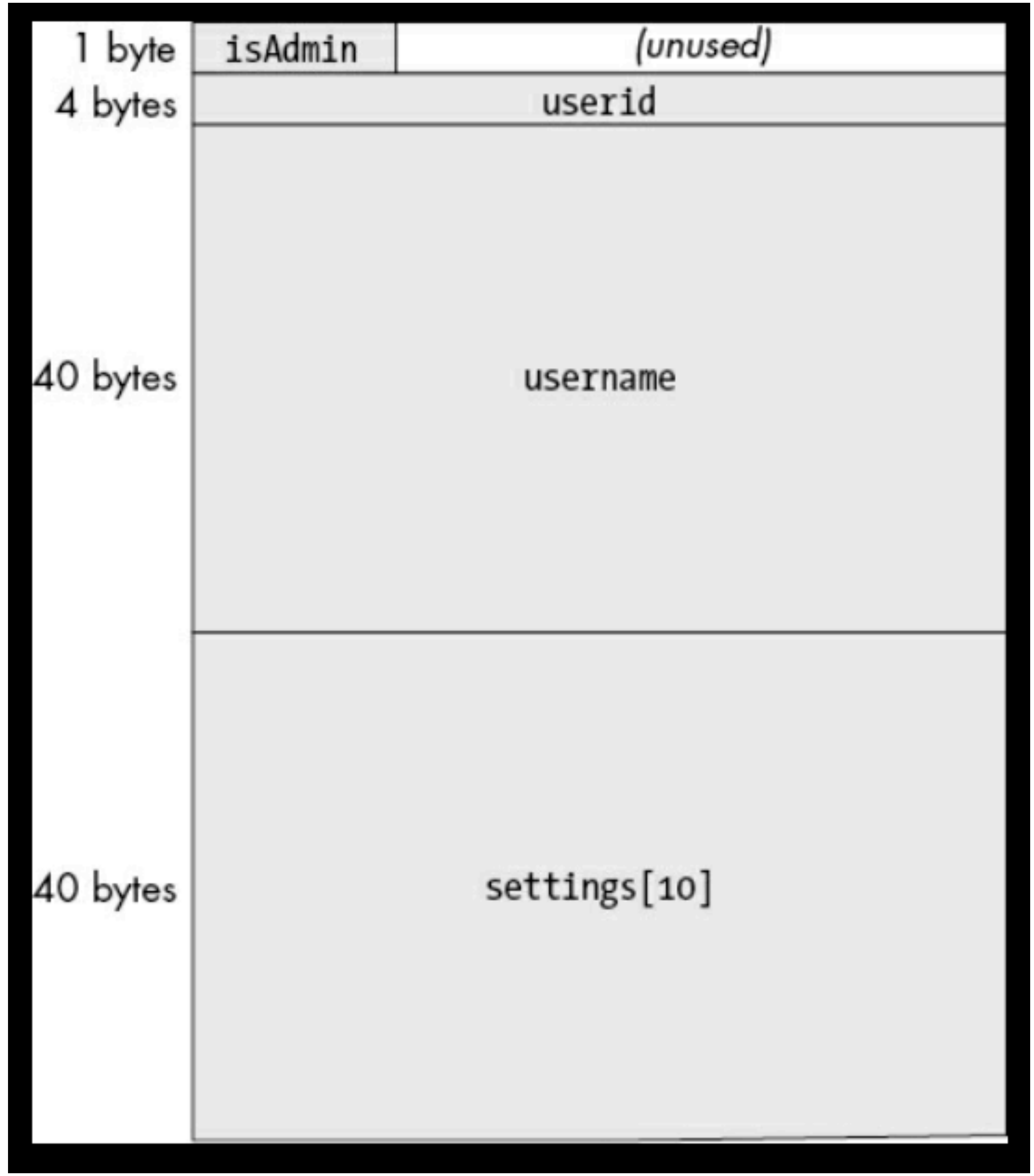
Example: Memory Allocation Vulnerabilities

- C data structure
- Writing data to **username - 12** can make someone an admin

```
#define MAX_USERNAME_LEN 39
#define SETTINGS_COUNT 10
typedef struct {
  bool isAdmin;
  long userid;
  char username[MAX_USERNAME_LEN + 1];
  long setting[SETTINGS_COUNT];
} user_account;
```

Leaking Memory

- **alloc()** does not initialize memory
- It contains leftover data from previous heap data
- This can leak out information
- **Mitigation**
- Write zeroes to whole data structure before use



strcpy

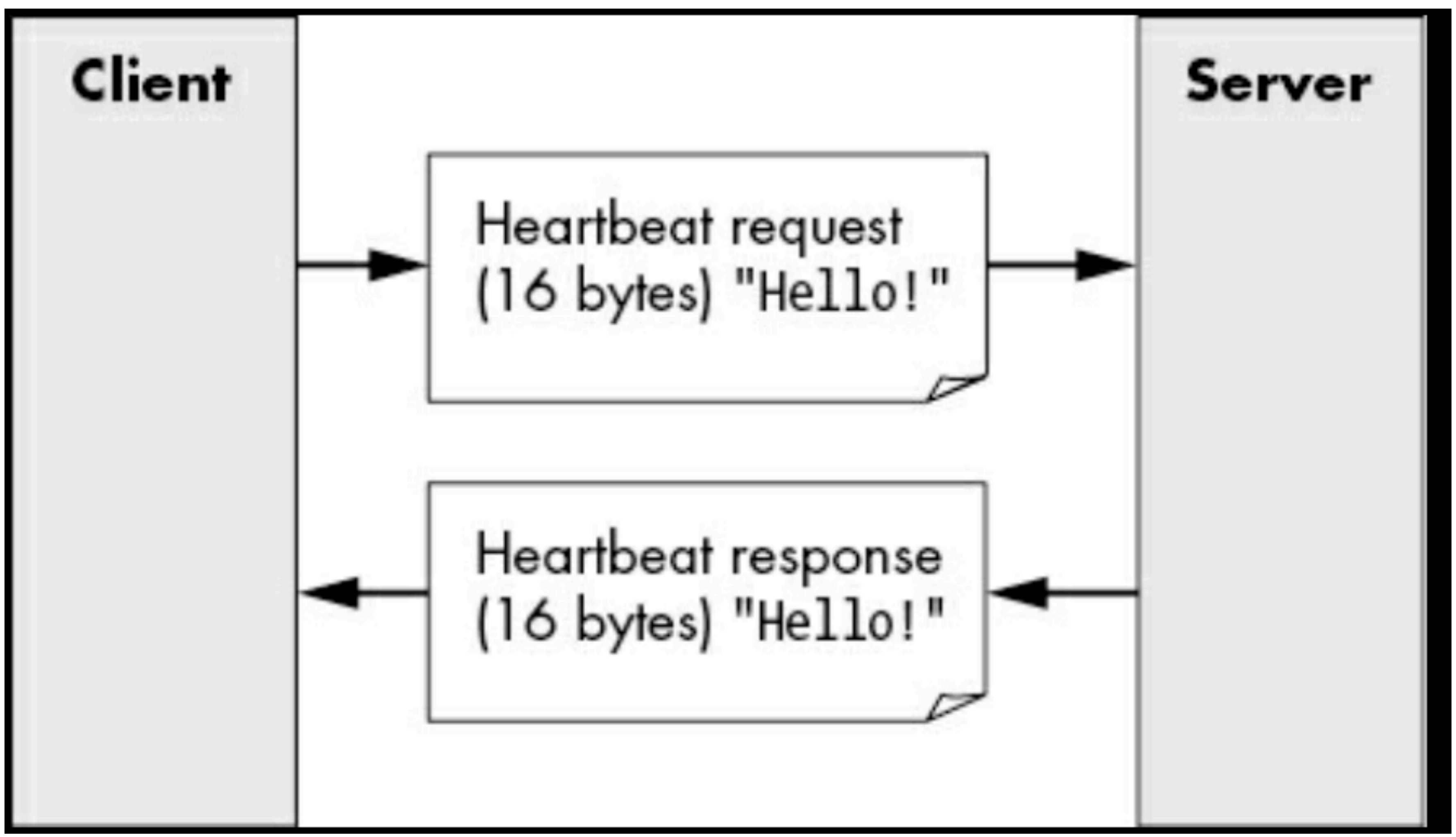
- Copies the source string, up to its null terminator
  - Into the destination string
- May copy a long string into a shorter one, overflowing its buffer
- May copy a short string into a longer one, leaving bytes uninitialized

strncpy

- Copies only a limited number of bytes from one string to another
- BUT does not ensure a null terminator in the destination string
- May expose data past the end of the destination string

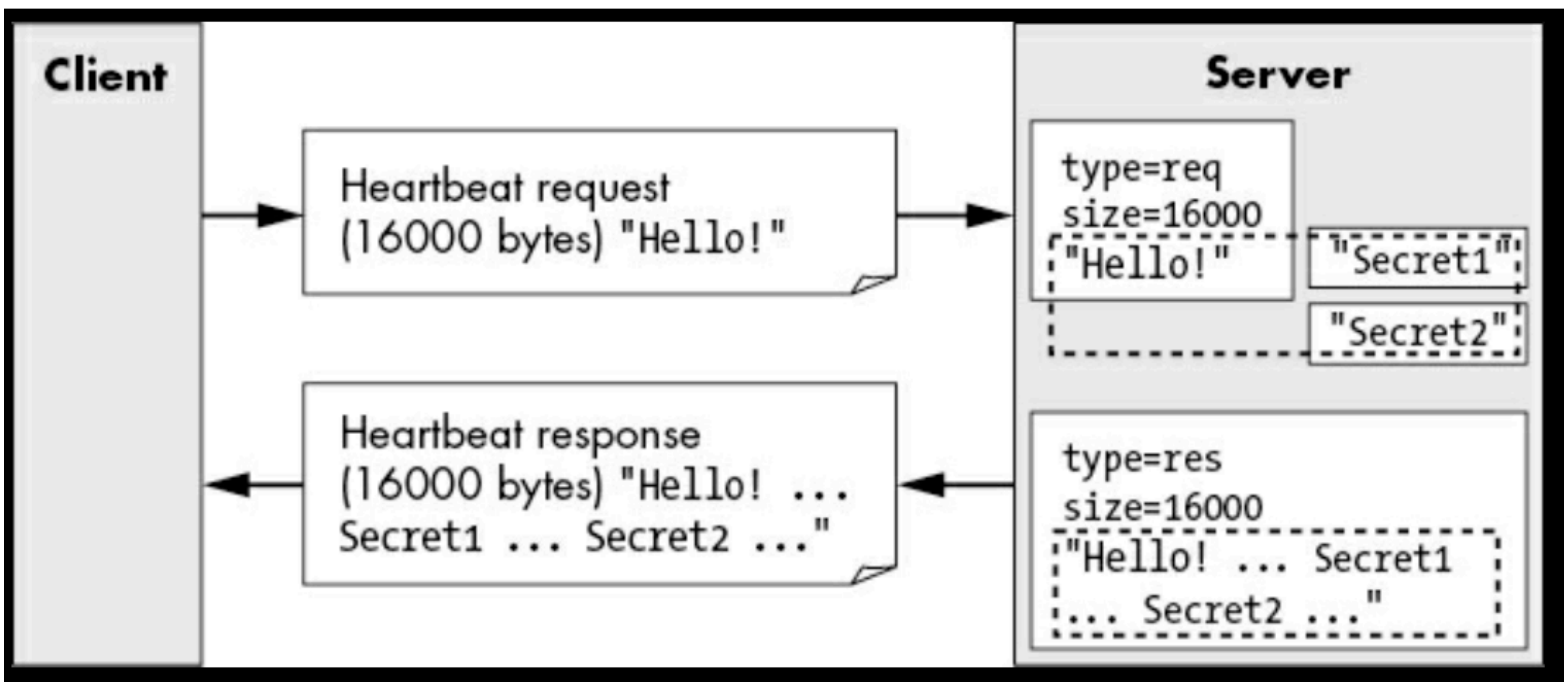
Case Study: Heartbleed

- Flaw in the openssl implementation of the TLS Heartbeat Extension
- Client sends a message to the server, which echoes it back



Case Study: Heartbleed

- BUT openssl trusted the length parameter in the request, without verifying it
- So send a short message with a long length
  - The server replies with a large chunk of uninitiaized memory



Remediation

- Compare the length parameter with the length of the actual data provided
  - If they are not the same, ignore the request

Kahoot!

Ch 9