# Secure Software Development

## Memory Corruption I (Print Version)

**Daniel Gruss, Vedad Hadzic, Martin Schwarzl, Samuel Weiser**

15.10.2020

Winter 2020/21, www.iaik.tugraz.at

# Table of contents

# Memory Safety

**Memory safety - Wikipedia**

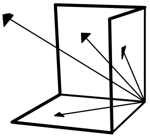
Memory safety is a concern in software development that aims to *avoid software bugs* that cause security *vulnerabilities* dealing with random-access memory *(RAM) access*, such as buffer overflows and dangling pointers.

# Memory Safety

A program execution is memory safe if the following things do not occur:

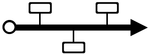
- access errors
  - buffer overflow/over-read
  - invalid pointer
  - race condition
  - use after free
- uninitialized variables
  - null pointer access
  - uninitialized pointer access
- memory leaks
  - stack/heap overflow
  - invalid free
  - unwanted aliasing

We can distinguish between two types of memory safety violation



Spatial violation: memory access is out of object's bounds

- buffer overflow
- out-of-bounds reads
- null pointer dereference



Temporal violation: memory access refers to an invalid object

- use after free
- double free
- use of uninitialized memory

# Memory Safety Violation

- Most “important” bugs are due to violation of memory safety
- Why can’t programming languages prevent them?
- There are memory safe languages (e.g., Rust, Java, ...), but...
  - ...most code is still written in C/C++
  - ...C/C++ is supported nearly everywhere
  - ...low-level code (e.g., operating systems) can’t easily be implemented in memory safe languages
  - ...memory safe languages are still not mature
- In which language is the runtime of a memory safe language written in?

# Memory Safety Violation Overview




Overflow (this lecture)

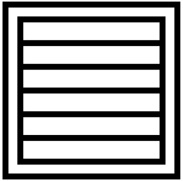
- Stack overflow
- Heap overflow
- Integer overflow


Invalid Memory (next lecture)

- Use-after-free
- Format string
- Type confusion

# Buffers

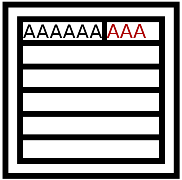
# Buffer



- A buffer is a chunk of memory...
  - with boundaries
  - defined by a start address and size
  - storing elements of a certain type
- Example: Arrays in C/C++

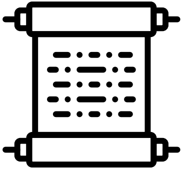
```
char buffer[12];
strcpy(buffer, "Hello");
```

# Buffer Overflow



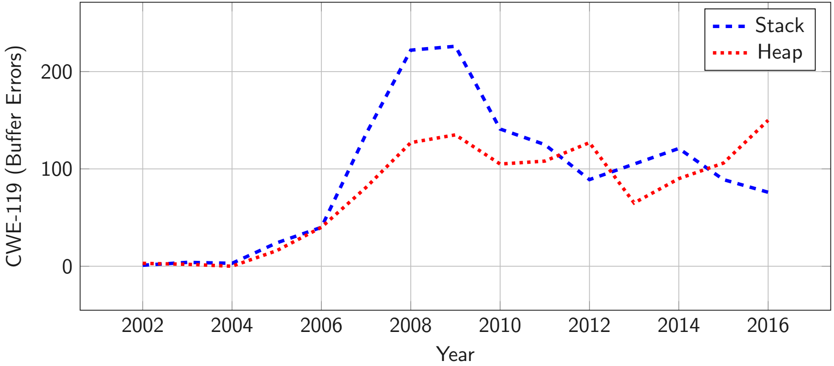

- Not all buffers check their bounds
- Out-of-bounds reads/writes access *something*
- Most commonly: array index out of bounds
- Example: Buffer overflow in C/C++

```
char buffer[4];
strcpy(buffer, "Hello");
```

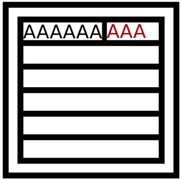


- 1972 First documentation of buffer overflows
- 1988 Morris Worm (aka “The Internet Worm”)
- 1996 AlephOne’s Phrack article “Smashing the Stack for Fun and Profit”
- 1998 DilDog’s tutorial “The Tao of Windows Buffer Overruns”
- 2000 Buffer overflows are “Bug of the decade” (beating Y2K bug)
- 2001 Halvar Flake predicted heap overflows to be the next wave
- 2002 Slapper infected Apache webservers using heap overflows
- 2003 Buffer overflows in Xbox games used to run unlicensed software
- ... A lot more buffer overflows

Buffer overflows are very common



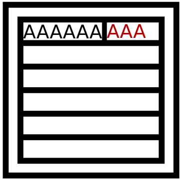
# Stack Overflow




- Local buffers are on the stack
- What is next to the buffer?
  - Other variables
  - Function parameters
  - Saved return addresses
- Attacker controls the buffer input, overwrites this data
- Changes control flow or manipulates data

# Buffer Overflow (Stack)

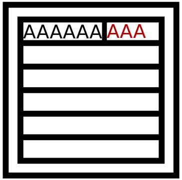




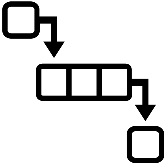

- Attacker can jump to arbitrary location in memory
- Every function that is mapped in the address space can be executed
- Attacker has effectively full control over the program

# Heap Overflow

# Buffer Overflow (Heap)




- Dynamic buffers (e.g., `malloc`'d) are on the heap
- What is next to the buffer?
  - Other variables
  - vtables of C++ objects
  - Internal data structures of `malloc`
- Attacker controls the buffer input, overwrites this data
- Changes control flow or manipulates data



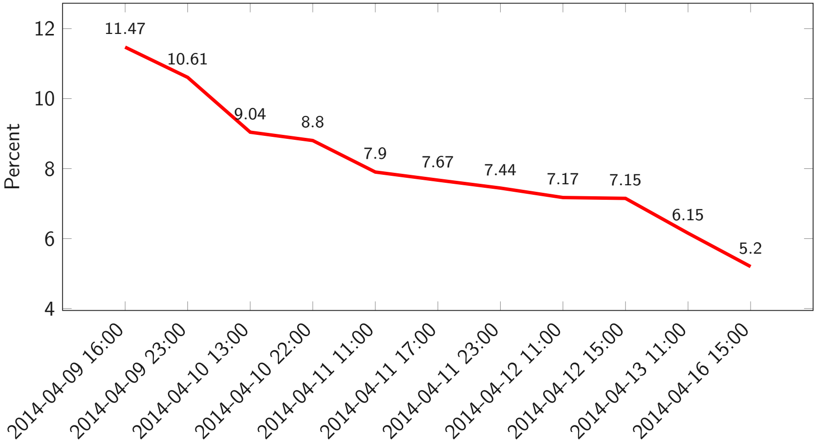
- Lots of different `malloc` implementations
  - `jemalloc` (Android, FreeBSD, Firefox)
  - `tcmalloc` (Chrome)
  - `dlmalloc/ptmalloc` (glibc)
- They all handle lists of chunks
- Chunks usually consist of meta data and user data
- There are various techniques to corrupt meta data to
  - achieve arbitrary memory reads/writes
  - get overlapping memory chunks

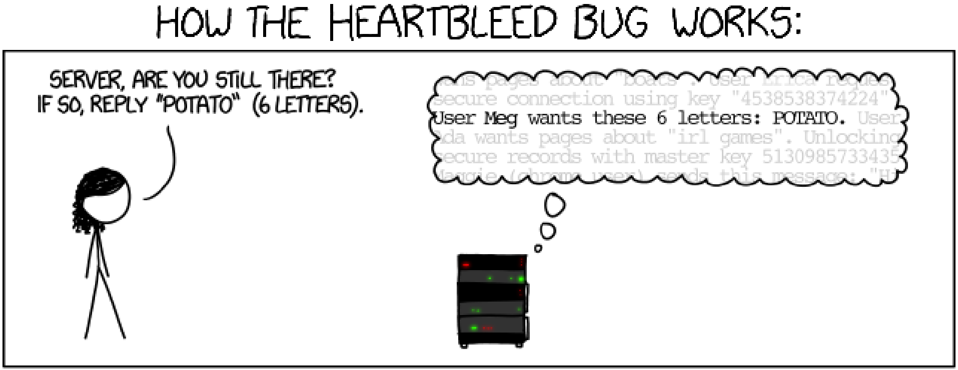
# Buffer Overflow (Heap) - vtable

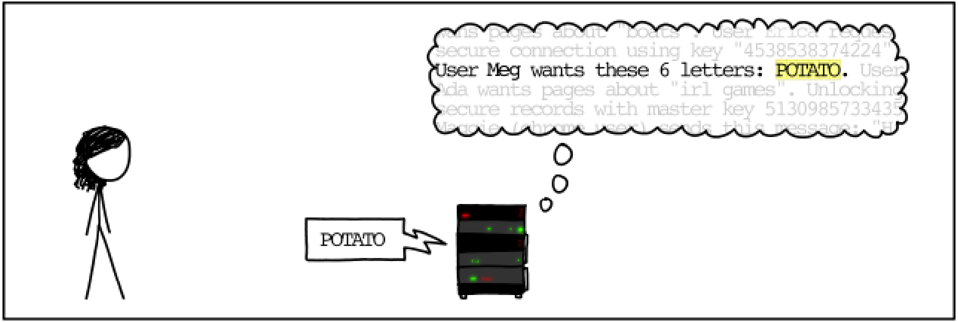
- C++ objects with virtual methods contain a pointer to a vtable
- The vtable contains function pointers
- If the buffer is before an object, we can overwrite the vtable pointer to an own, crafted vtable
- Controlling the vtable pointers allows to call arbitrary functions

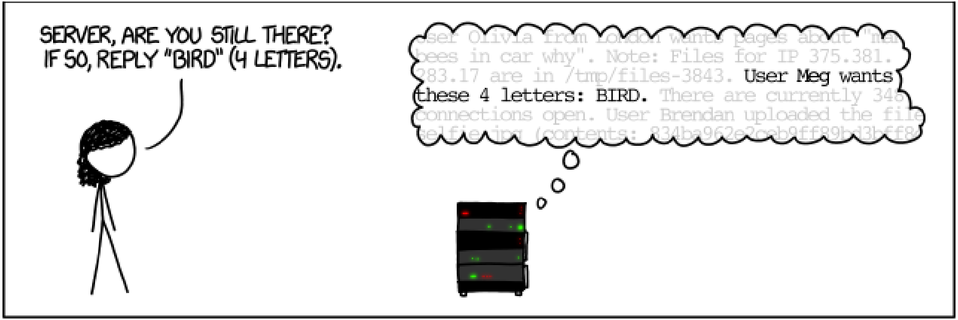
- A security bug in the TLS protocol implementation of OpenSSL
- In the *heartbeat* extension (hence the name)
- A missing bounds check leads to a buffer over-read
- Allows to read up to 64 KB of server memory

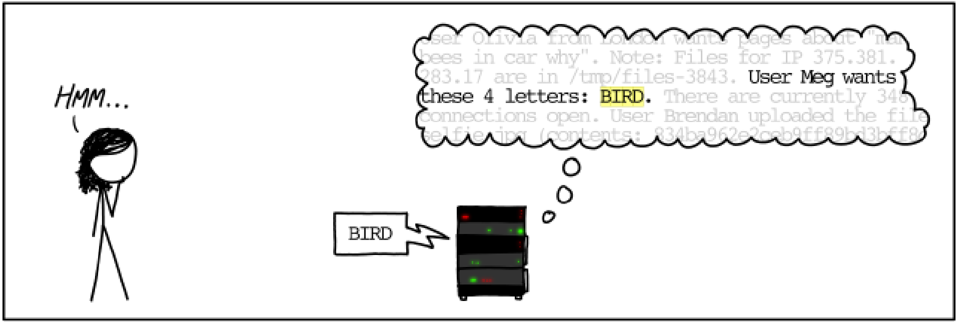
Alexa Top 1 Million Pages - Vulnerable servers

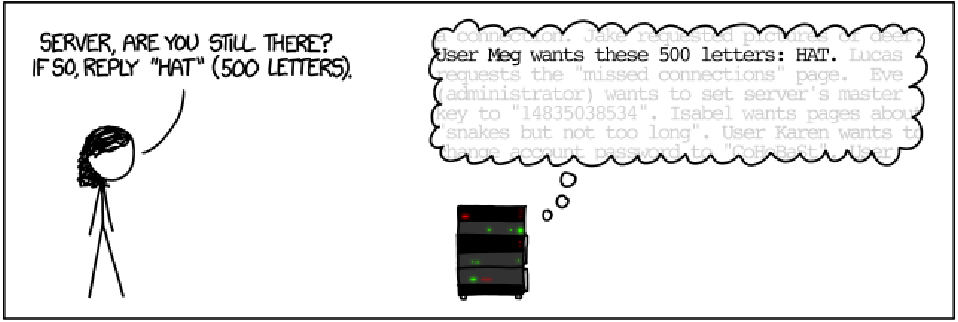


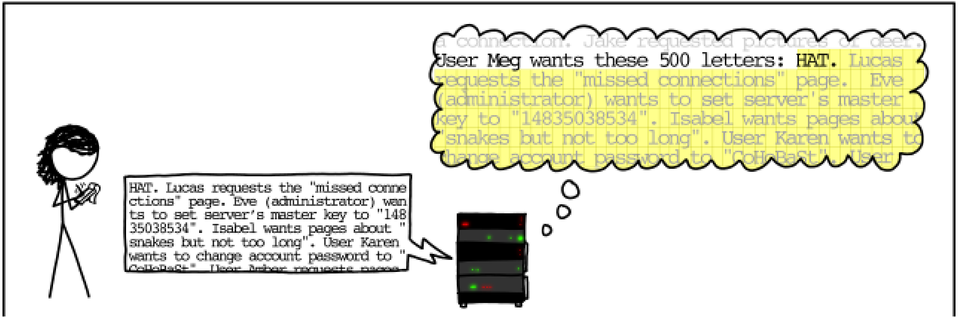

HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
secure connection using key "4538538374224".
User Meg wants these 6 letters: POTATO. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435


secure connection using key "4538538374224"
User Meg wants these 6 letters: POTATO. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435
POTATO


SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Olivia from London wants pages about "ma
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 348
connections open. User Brendan uploaded the file


HMM...
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 346
connections open. User Brendan uploaded the file
BIRD


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages abou
"snakes but not too long". User Karen wants to


HAT. Lucas requests the "missed conne
ctions" page. Eve (administrator) wan
ts to set server's master key to "148
35038534". Isabel wants pages about "
snakes but not too long". User Karen
wants to change account password to "
User Meg wants these 500 letters: HAT. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
"snakes but not too long". User Karen wants to

# Heartbleed Code

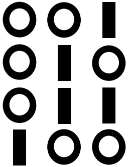
```
struct
{
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;

/* Read type and payload length first */
hbtype = *p++; // message type
n2s( p ,  payload ); // payload = received payload length
pl = p; // pl = content of payload

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE; // message type
s2n( payload , bp); // payload length to message (bp)
memcpy(bp, pl,  payload ); // copy payload bytes from original content to message
```

- Evil C functions for string handling (`gets`, `strcpy`, ...)
- Off-by-one errors (Null-Byte BOFs)
- Unicode vs ANSI (different size for characters)
- Wrong loop termination (e.g., off-by-one)
- Arithmetic errors (e.g., integer overflows)

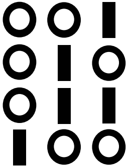

# Integers



- There are different formats for storing numbers
- Binary for unsigned integers, only positive numbers
- Two's complement for signed integers, positive and negative
- Sign bit + Magnitude for floating point numbers

# Numbers in Memory - Unsigned Integers





- An $n$-bit integer $x$ is represented as

$$x = (x_{n-1}, x_{n-2}, \cdots, x_1, x_0) = \sum_{i=0}^{n-1} 2^i \cdot x_i$$

- The range of representable values is

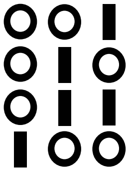
$$0 \leq x < 2^n$$

- On overflow, the value is reduced modulo $2^n$

$$x = \begin{cases} x & x < 2^n \\ x \mod 2^n & x \geq 2^n \end{cases}$$

# Numbers in Memory - Signed Integers





- An $n$-bit integer $x$ is represented as

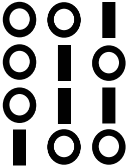
$$x = (x_{n-1}, x_{n-2}, \cdots, x_1, x_0) = -2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} 2^i \cdot x_i$$

- The range of representable values is

$$-2^{n-1} \leq x < 2^{n-1}$$

- Two's complement has a negate operation

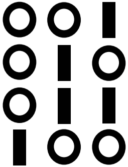
$$-x = 2^n - x$$



- A single-precision (IEEE 754-2008) float $x$ is represented as

$$\begin{aligned} x &= (x_{31}, x_{30}, \cdots, x_1, x_0) \\ &= (-1)^{x_{31}} \cdot \left(1 + \sum_{i=1}^{23} x_{23-i} 2^{-i}\right) \cdot 2^{([x_{30}:x_{23}]-127)} \end{aligned}$$

- A single-precision float can encode numbers up to $\approx 3.4 \times 10^{38}$
- All integers with $\leq 6$ decimal digits can be encoded
- All values $2^n$ with $-126 \leq n \leq 127$ can be encoded
- Compact: 1 bit (sign), 8 bit (exponent), 23 bit (fraction/mantissa), bias=127 (since stored as unsigned)



- Example: $x = 3.3125 = 11.0101_b$
- Normalize to $1.bbb \times 2^e = 1.10101_b \times 2^1$
- Sign bit: 0 as it is positive
- Exponent: $e + 127 = 1 + 127 = 128$
- Fraction: $0.bbb \times 2^{23}$
  $= 0.10101_b \times 2^{23} = 0.65625 \times 2^{23} = 5505024$
- Result: $01000000010101000000000000000000_b$

```c
int i = 0b01000000010101000000000000000000;
float f = *(float*)&i;
printf("%.4f\n", f); // prints 3.3125
```

# Challenge #2 - Quadruple-precision floating point



**Given the number "`-135253521335.224627`", convert it to IEEE 754 quadruple-precision binary floating-point format (binary128)**

- The solution is the decimal interpretation of the fraction part (cf. lecture slides example)
- Example: 9876543210.5 has the decimal interpretation of the fraction part 7777071893216791224292541233888928
- You can do it manually or use any program you like
- Format description: `https://en.wikipedia.org/wiki/Quadruple-precision_floating-point_format`

```c
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  i  = * ( long * ) &y;                     // evil floating point bit level hacking
  i  = 0x5f3759df - ( i >> 1 );             // what the fuck?
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
  // y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, can be removed

  return y;
}
```

- The infamous fast inverse square root from Quake III Arena
- Computes $\frac{1}{\sqrt{x}}$ with quite good precision
- Origins of the "hack" not fully known
- Also unknown how the magic number 0x5F3759DF was found
- Abusing low-level representation led to algorithm four times faster than all other algorithms

# Integer Overflow

- What happens on an overflow?

**Paragraph 5/4, C++11 Standard**

If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

This applies only to signed integers, because

**Paragraph 3.9.1/4, C++11 Standard**

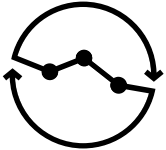
Unsigned integers, declared unsigned, shall obey the laws of arithmetic modulo $2^n$ where n is the number of bits in the value representation of that particular size of integer [...] unsigned arithmetic does not overflow because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type.

- An unsigned $n$-bit integer can overflow in multiple cases
- Addition: $a + b \geq 2^n \quad (0 \leq a, b < 2^n)$
- Subtraction: $a - b < 0 \quad \text{if } b > a \quad (0 \leq a, b < 2^n)$
- Multiplication: $a \cdot b \geq 2^n \quad (0 \leq a, b < 2^n)$

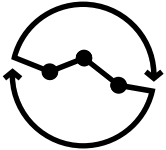
- A signed $n$-bit integer can overflow in multiple cases
- Addition/Subtraction: $a + b \geq 2^{n-1}$ or $a + b < -2^{n-1}$ $(-2^{n-1} \leq a, b < 2^{n-1})$
- Negation: $a = -2^{n-1} \Rightarrow -a = 2^{n-1}$
  "Asymmetry" of two's complement
- Multiplication: $a \cdot b \geq 2^{n-1}$ or $a \cdot b < -2^{n-1}$ $(-2^{n-1} \leq a, b < 2^{n-1})$

  Multiplication by $-1 \Rightarrow$ Negation
- Division: $\frac{-2^{n-1}}{-1} = 2^{n-1} \Rightarrow$ Negation

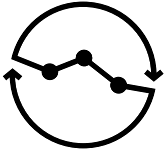
  Division by 0



- C has rules to automatically convert types (coercion)
- Type conversion is done by the compiler and can have unintended consequences
  - `float` to `int` causes truncation (removal of the fractional part)
  - `double` to `float` causes rounding of digit
- Similar to type conversion, there is conversion from smaller to larger data types



- Converting smaller to larger data types can be done using
  - Sign extension (high bits are set to the *sign* bit) or
  - Zero extension (high bits are set to '0's)
- If an assignment has two
  - Signed integers ⇒ sign extension
  - Unsigned integers ⇒ zero extension
  - Mixed integers ⇒ it depends...
    - Zero extension if source is unsigned
    - Sign extension if source is signed



Type conversion for arithmetic operations

- Same type, same rank†: no conversion
- Same type, different rank: convert smaller to larger data type
- Different type: complicated...
  - unsigned integer has same or higher rank than signed integer ⇒ convert to unsigned
  - otherwise, signed integer can represent unsigned integer ⇒ convert to signed
  - otherwise, convert both operands unsigned with type of signed integer

† rank is similar to size, an integer contains at least as many bits as the types ranked below it

- Integer overflows are not a memory safety violation on their own
- They can lead to a memory safety violation if used...
  - for pointer arithmetic
  - as `malloc` argument
  - as array index
- Lead often to buffer overflows
- Can also result in out-of-bounds read/write

# Real-world Integer Overflow



```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];
        if (midVal < key)
            low = mid + 1
         else if (midVal > key)
            high = mid - 1;
         else
            return mid; // key found
     }
     return -(low + 1);  // key not found.
}
```

# Fixing Integer Overflow



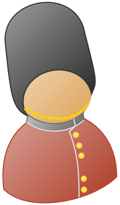
```
void* new_8bit_image(unsigned int width, unsigned int height)
{
    unsigned int memory = width * height;
    void* data = malloc(memory);
    return data;
}
```

# Fixing Integer Overflow

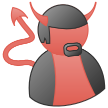
```
void* new_8bit_image(unsigned int width, unsigned int height)
{
    unsigned int memory = width * height;
    if(width * height > UINT_MAX) return NULL;
    void* data = malloc(memory);
    return data;
}
```

# Fixing Integer Overflow



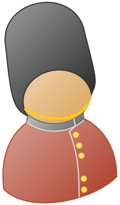


```
void* new_8bit_image(unsigned int width, unsigned int height)
{
    unsigned int memory = width * height;
    if(UINT_MAX / width < height) return NULL;
    void* data = malloc(memory);
    return data;
}
```

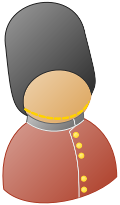


What if `width` == 0?

# Fixing Integer Overflow



```c
void* new_8bit_image(unsigned int width, unsigned int height)
{
    unsigned int memory = width * height;
    if(!width || (UINT_MAX / width < height)) return NULL;
    void* data = malloc(memory);
    return data;
}
```



```
void* new_8bit_image(unsigned int width, unsigned int height)
{
    unsigned int memory;
    if(__builtin_umul_overflow(width, height, &memory)) {
      return NULL;
    }
    void* data = malloc(memory);
    return data;
}
```

- GCC/clang provide built-in functions to check for overflows
- `__builtin_add_overflow`, `__builtin_sub_overflow`, `__builtin_mul_overflow` for various data types

# Overflow Summary



Overflows...

- are the most common forms of memory safety violation
- are mostly caused by missing bound checks
- can be abused to read from and write to memory
- might occur on buffers and integers
- exist in nearly every programming language (some exceptions)

- Will Dietz, Peng Li, John Regehr, and Vikram Adve.
  **Understanding integer overflow in C/C++.**
  *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2015.
- Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al.
  **The matter of heartbleed.**
  In *Proceedings of the 2014 Conference on Internet Measurement Conference*, 2014.
- Aleph One.
  **Smashing the stack for fun and profit.**
  *Phrack magazine*, 7(49), 1996.
- Ahmad-Reza Sadeghi.
  **Secure, Trusted and Trustworthy Computing (TU Darmstadt).**

sploitfun.
**Understanding glibc malloc.**

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song.
**Sok: Eternal war in memory.**
In *Security and Privacy (SP), 2013 IEEE Symposium on*, 2013.