# Abstraction example - Integers

# Variable semantics

- What does the following line of code do?

```
unsigned k = 0;
```

- Allocates space
- Associates the space with a
- Defines how the preted
- Initialises the sto

Let's t this

# Example

```c
#include <stdio.h>

int main() {
  int i;
  int sum = 0;

  for (i = 0; i < 10; i++)
    sum += i;

  printf("The sum is %d\n", sum);
}
```

```
_main:
    pushq    %rbp
    movq     %rsp, %rbp
    L_.str(%rip), %rdi
    $45, %esi
             %eax, %eax
            _printf
    xorl     %eax, %eax
    popq     %rbp
    retq
L_.str:
    .asciz   "The sum is %d\n"
```

## Compiler abstractions

- The compiler generates a ***concrete implementation*** of an ***abstract program*** description
- Preserves semantics (to some extent)
- Changes imple
  - May not match intuition
- We need to understand the abstraction

# Unsigned integers

- What is the output from:

```
$ count 4
$ count 294
$ count 67295
```

```
#include <stdio.h>
#include <stdlib.h>

int main(int c, char **v) {
           i, n;

               (v[1], NULL, 0);

  for (i = n; i < n+10; i++)
    printf("%u\n", i);
}
```

## Unsigned integer representation

- Collection of $n$ bits $b_0 \ldots b_{n-1}$
- Represents the number
- $n$ depends on t e implementation
- An overflow occurs when a result is larger than $2^n$
  - All arithmetic is modulo $2^n$

# Signed Integers

- One sign bit $s$ and $n$-1 value bits $b_0 \ldots b_{n-2}$
- Three possible interpretations:
  - Sign magnitude
  - Ones' complem
  - Two's complement
- Most modern processors use two's complement

# Signed integer overflow

- Undefined behaviour
  - Use modulo $2^{n-1}$ arithmetic
  - Return maximum or minimum values
  - Return zero
  - Do nothing
  - Cause a trap
  - Launch $500M fireworks

## Integer overflow vulnerabilities

## Stagefright

- Before

Why not use
`chunk_size+size >= SIZE_MAX` ?

- After

# Type Conversion

```
bool isValidAddition(uint16_t x, uint16_t y)
{
  if (x + y < x)
    return false
  return true;
}
```

```
if ((uint16_t)(x + y) < x)
```

## CVE-2017-7602 (LibTIFF)

ma is positive

mb >= size
(overflow ignored)

⋮

## Fix: test for overflow

## Best practices

- Know the language
  - Undefined behaviours are dangerous

- Test user input
  - Special attentio cation

- Use safe tests
  - Subtract from maximum
- Use explicit casts when using types smaller than `int`

## Language Support

- Java:
  - **`Math.multiplyExact`**, **`Math.addExact`**, etc.

- C/C++ compiler
  - **`-fwrapv`**, **`-ft`**

  - **`-fsanitize`**

- C#
  - **`checked`**