# My Behaviors Are Undefined

LSE
Security System
Laboratory of Epita

Why You Should Read the Docs

## Overview

- Sequence Points
- Integer Madness
- Floating Points Approximations
- libc (ISO/POSIX) Real Behavior
- Standard version ?
- ...

LSE Security System
Laboratory of Epita

## C Semantics

**Standard defines several situations:**

- ➢ Well defined behavior
- ➢ Implementation dependant behavior
- ➢ Undefined behavior

## Common Issues

- ➢ Sequence points and execution order
- ➢ Signed integer overflow
- ➢ Cast and conversion
- ➢ Data alignment
- ➢ ...

LSE Security System Laboratory of Epita

## Sequence Points

## Execution Order ?

```c
int next() {
  static int x = 0; return x++;
}
void g() {
  int x = 0, y, tab[32];
  // can be equivalent to:
  // tab[0] = 1
  // tab[1] = 0;
  // ...
  tab[x++] = x++;
  // x = 2 - 1 or 1 - 1 ?
  y = x + --x;
  // x = 0 - 1 or 1 - 0 ?
  x = next() - next();
}
```

LSE Security System Laboratory of Epita

## Sequence Points ?

Between 2 **sequence points**, all subexpressions can be reordered by the compiler.

**From the standard:**

*At certain specified points in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place. (§1.9/7)*

LSE Security System
Laboratory of Epita

## Sequence Points in C

- At the end of a full expression (§1.9/16)
- At function calls (after arguments) (§1.9/17)
- In the following expressions after the evaluation of the first one (§1.9/18):
  - a && b (§5.14)
  - a || b (§5.15)
  - a ? b : c (§5.16)
  - a , b ($5.18)

LSE Security System
Laboratory of Epita

## Concrete Example

```c
#include <stdio.h>

int x = 0;
int main() {
  printf("%d ", (x = 3) + (x = 4));
  printf("%d\n", x);
  return 0;
}
```

Expected output:

- 7 3
- 7 4

gcc ouput:

- 7 4 (-O0)
- 8 4 (-O1)

clang ouput:

- 7 4 (-O0)
- 7 4 (-O1)

Security System
Laboratory of Epita

## Integer Madness

## Integer Overflow

```
void sample(int a, int b) {
  if (a > 0 && b > 0) {
    if (a + b < 0)                  // Wrong
      warnx("Overflow");
    if (INT_MAX - a < b)            // Good !
      warnx("Overflow");
  }
}
```

- ➢ The test `(a + b < 0)` is false or undefined
- ➢ Compilers may simply delete the block

## Is There A Problem ?

```c
#include <stdio.h>

int main() {
  int si = -1;
  unsigned int ui = 1;
  printf("%d\n", si < ui);
  return 0;
}
```

What do you think ?

- ➢ Print 1 ?
- ➢ Print 0 ?

LSE Security System
Laboratory of Epita

## Yes ...

The previous code print ... **0**

**Why ?**

Have you heard about integer promotion ?

## Ranks And Conversions

Integer types are subject to a complex set of conversion rules.

Types are ranked depending on their signedness and size.

In many cases conversion are not *natural* and may yield unexpected results.

LSE Security System
Laboratory of Epita

## Another Example

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

int main() {
  uint8_t port = 0x5a;
  uint8_t result_8 = (~port)>>4;
  printf("0x%hhx\n", result_8);
  return 0;
}
```

Expected value:

➢ 0x0a

Obtained value:

➢ 0xfa

LSE Security System
Laboratory of Epita

# Explanations

| Expression | Type | Value |
|---|---|---|
| `port` | `uint8_t` | `0x5a` |
| `~port` | `int` | `0xffffffa5` |
| `(~port)>>4` | `int` | `0xfffffffa` |

Security System
Laboratory of Epita

## Solutions ?

```c
int si = -1;
unsigned int ui = 1;
printf("%d\n", (si < 0) || si < (int)ui);
```

```c
uint8_t port = 0x5a;
uint8_t result_8 = ( (uint8_t)~port ) >> 4;
```

LSE Security System
Laboratory of Epita

## Floating Points

LSE Security System

Laboratory of Epita

# Absorbing

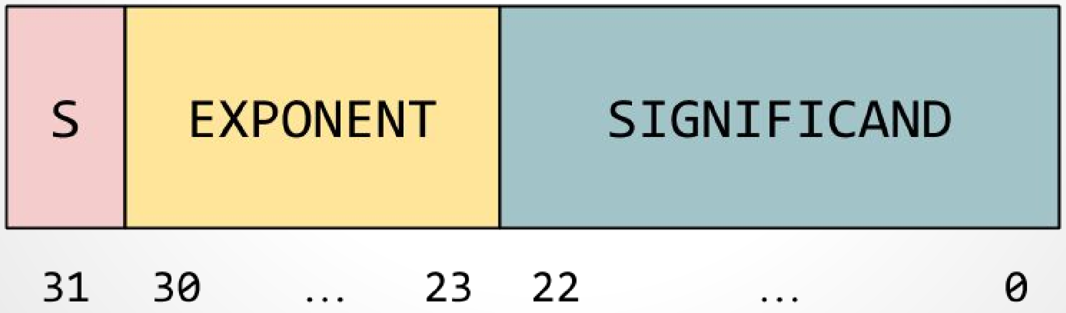
```c
float showing_absorb(size_t len) {
  volatile float          r = 0;
  for (size_t i = 0; i < len; ++i)
    r += 1;
  return r;
}
int main() {
  size_t                  i = 0;
  float                   r;
  do {
    i += 1;
    r = showing_absorb(1<<i);
  } while ((float)(1<<i) <= r);

  printf("get out with: \t2^%zu\n", i);
  printf("len:\t\t%zu\n", (size_t)1<<i);
  printf("expected: \t%g\n",
(float)(1<<i));
  printf("result: \t%g\n", r);
  return 0;
}
```

```
get out with: 2^25
len:          33554432
expected:     3.35544e+07
result:     1.67772e+07
```

## Floating Points Number

- Form: (+/-) `1.M * 2`$^{(e\ -\ bias)}$
- M: mantissa (23bits)
- e: exponent (8bits)
- Special values: NaN, infinity ...



# Float Vector Sum

*Divide and Conquer principle:*

```
➢ sum(vect[m,m+1[)  → vect[m]
➢ sum(vect[m,M[)    →
        sum(vect[m, m + (M-m)/2[)
      + sum(vect[m + (M-m)/2, M[)
```

LSE Security System
Laboratory of Epita

## More

- ➢ Order in computation matters
- ➢ Compiler’s optimization preserves semantics
- ➢ Option `-ffast-math` break semantics
- ➢ Readings:
  - Comparing Floating Point Numbers
  - Floating-Point Determinism
  - Beginner's Corner: Floating Point Numbers

## Standard And Behavior

## Standard And Behavior

A common **false** assumption is to think that a piece of code will have the same behavior when compiled with different versions of the standard.

## Simple Code Example

```c
/* a.c */

#include <stdio.h>

int bf(unsigned short a) {
  printf("> 0x%hx\n", a);
  return 0;
}
```

```c
/* main.c */

#include <stdio.h>
#include <unistd.h>

int main(void) {
  unsigned long x;
  x = 0x12345678;
  bf(0x12345678);
  printf(": 0x%lx\n", x);
}
```

LSE Security System
Laboratory of Epita

## Simple Code Example

This code is valid in ISO C90 (ANSI C89)

It even runs almost correctly !

```
sh> cc -std=c89 -O0  -m32 -c -o main.o main.c
sh> cc -std=c89 -O0  -m32 -c -o a.o a.c
sh> cc  -m32 main.o a.o   -o main
sh> ./main
> 0x5678
: 0x12345678
sh> echo $status
13
```

## Implicit Functions & Return

Supposing C90/C89 standard:

- ➢ Undeclared function F is seen as **`int F()`**
- ➢ Parameters are passed as `int`
- ➢ Return value, if missing, is the last value on the stack (here the result of printf.)

## Simple Code Example

```
sh> cc -std=c99 -O0  -m32 -c -o main.o main.c
main.c: In function 'main':
main.c:11: warning: implicit declaration of function 'bf'
sh> cc -std=c99 -O0  -m32 -c -o a.o a.c
sh> cc  -m32 main.o a.o   -o main
sh> ./main
> 0x5678
: 0x12345678
sh> echo $status
0
```

## More On Implicit Declarations

Implicit return type (int) of implicitly declared functions can leads to strange behavior:

```
/* foo.c */
#include <stdio.h>
int main() {
  int *p;
  p = (int*)malloc(4);
  printf("> %p\n",p);
  *p = 42;
}
```

```
sh> cc -std=c89 -fno-builtin foo.c
…
sh> ./a.out
> 0xc07090
Segmentation fault
```

## More On Implicit Declarations

Implicit return type (int) of implicitly declared functions can leads to strange behavior:

```c
/* foo.c */
#include <stdio.h>
int main() {
  int *p;
  p = (int*)malloc(4);
  printf("> %p\n",p);
  *p = 42;
}
```

```
sh> cc -m32 -std=c89 -fno-builtin foo.c
sh> ./a.out
> 0x28404088
sh>
```

## More On Implicit Declarations

In the previous slides, I’ve hidden the compiler's warnings … (*What !*)

What’s the deal ? On a 64bits machine, `int` are 4 bytes long but pointers are 8 bytes long …

## LibC Behavior

## Slow Inputs

When using read(2) an output value of -1 normally indicates an error … *does it* ?

There's a notion of slow inputs: file descriptors such that data may not be available directly.

LSE Security System

Laboratory of Epita

## EINTR and EAGAIN

The standard specifies that when a signal arrives while the process is waiting data from a slow input, read(2) should fail with errno(3) set to EINTR.

It also states that when reading on an non-blocking input (if no data are available), read(2) will fail with errno(3) set to EAGAIN or EWOULDBLOCK

## Correct reading loop

```c
void cat(int fdin, int fdout) {
  int           r;
  char          buf[1024];
  while ( (r = read(fdin, buf, 1024)) ) {
    if (r == -1) {
      if (errno == EINTR
          || errno == EAGAIN
          || errno == EWOULDBLOCK )
        continue;
      err(3, "problem while reading ...");
    }
    r = write(fdout, buf, r);
    if ( r == -1 )
      err(3, "problem while writing ...");
  }
}
```

LSE Security System
Laboratory of Epita

## Really ?

Some OS provides automatic restart of syscall

« What ? My fd are slow ? NO »

**So do we need to test errno(3) ?**

LSE Security System Laboratory of Epita

## Yes We Need It !

Automatic restart on signal is not the standard behavior nor is it mandatory.

POSIX.1-2001 is ambiguous about the error for non-blocking socket

Every time you read on standard input you can be redirected !

## There's More …

- ➢ `pipe(2)` syscall on modern OS returns bidirectional fd but that's not standard
- ➢ R/W streams (stdio's `FILE*`) require seek between read and write operations
- ➢ And what's about `setcontext(2)` interaction with threads (anyway, `ucontext` operations are deprecated.)

LSE Security System

Laboratory of Epita

## Why Should I Care ?

LSE Security System

Laboratory of Epita

## You Must Care !

- C and C++ are the most used languages
- Most security issues come from code error
- Even if “*it’s working*” now, compilers evolve
- Most of you learn using x86, what about ARM

## “I don’t use C”

- Most points apply to any compiled language
- Higher-level languages show more complex undefined patterns.
- And Dynamic languages ?
  - Wrong code may run for some cases
  - Auto-conversion yield strange behavior (true=false)
  - VM changes can break a lot of stuff

LSE Security System
Laboratory of Epita