On Undefined Behavior in C/C++/Rust

Godmar Back

Virginia Tech

February 15, 2022

VIRGINIA TECH

An slightly modified example of CS3214 p3 student code

mm_malloc.c

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

#define NUM_LISTS   2
static int list_bsizes[NUM_LISTS] = { 0, 1024 };

static void find_fit(size_t asize) {
    int i = 0;
    for (; i < NUM_LISTS; i++) {
        if (asize <= list_bsizes[i + 1]) {
            return;
        }
    }
    printf("I'm here\n");
    return;
}
```

(cont'd)

```c
void *mm_malloc(size_t size) {
    if (size <= 1024)
        abort();

    find_fit(size);
    return NULL;
}
```

What happens if `mm_malloc(1025)` is called?

VIRGINIA TECH

An slightly modified example of CS3214 p3 student code

mm_malloc.c

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

#define NUM_LISTS   2
static int list_bsizes[NUM_LISTS] = { 0, 1024 };

static void find_fit(size_t asize) {
    int i = 0;
    for (; i < NUM_LISTS; i++) {
        if (asize <= list_bsizes[i + 1]) {
            return;
        }
    }
    printf("I'm here\n");
    return;
}
```

(cont'd)

```c
void *mm_malloc(size_t size) {
    if (size <= 1024)
        abort();

    find_fit(size);
    return NULL;
}
```

gcc 6.4, with -O2

```
mm_malloc(unsigned long):
    subq    $8, %rsp
    call    abort
```

VIRGINIA TECH

What happened?

- If `mm_malloc` is called with a size less or equal to 1024, it would call `abort()` which won't return.
- So `find_fit(size)` is called with a size greater than 1024.
- The first loop then will access `list_bsizes[2]` which is out of bounds.
- This is undefined behavior, which can't happen.
- Therefore, `find_fit()` isn't called.
- Therefore, `size` must have been less or equal to 1024.
- Therefore, `mm_malloc()` will always call abort (!?)

VIRGINIA TECH

Undefined vs. Unspecified Behavior

ISO/IEC 9899:2017: undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements[a]

[a]Note 1 to entry: Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

unspecified behavior

use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

implemented-defined behavior: unspecified behavior where each implementation documents how the choice is made

Examples of Unspecified Behavior

- The manner and timing of static initialization
- Many aspects of the representations of types
- The value of padding bytes when storing values in structures or unions
- Whether two string literals result in distinct arrays
- The order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call
- The order in which the operands of an assignment operator are evaluated
- ... and many more (total over 60 items)

VIRGINIA TECH

Examples of Undefined Behavior

- A "shall" or "shall not" requirement that appears outside of a constraint is violated[1]
- An object is referred to outside of its lifetime
- The value of a pointer to an object whose lifetime has ended is used
- The value of an object with automatic storage duration is used while it is indeterminate
- Conversion to or from an integer type produces a value outside the range that can be represented
- The execution of a program contains a data race
- ... the entire list fills 10 pages in appendix J.2

[1] The word "shall" appears 943 times in the document

VIRGINIA TECH

Security Impact

- In a paper at SOSP'13 [2], Wang et al analyzed code bases for reliance on undefined behavior. They found instances in thousands of C/C++ packages.

Source: Vulnerability Note VU#162289, US-CERT, 2008

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
  return; /* len too large */
if (buf + len < buf)
  return; /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

VIRGINIA TECH

Security Impact

- In a paper at SOSP'13 [2], Wang et al analyzed code bases for reliance on undefined behavior. They found instances in thousands of C/C++ packages.

Source: Vulnerability Note VU#162289, US-CERT, 2008

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
  return; /* len too large */
if (buf + len < buf)
  return; /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

- Compiler knows that pointer arithmetic overflow is undefined behavior, so it "knows" that `buf + len` is always greater or equal than `buf` and elides the check.

VIRGINIA TECH

Practical Consequences

- First, intuition is wrong: cannot reason about what might happen to code that exhibits undefined behavior. "It will crash..." or "It'll eventually overrun the bounds..." etc. do not work.
- Second, you must use the (limited) tools we do have available:
  - Valgrind: flags undefined behavior that the compiler didn't recognize and eliminate[2]
  - Undefined sanitizer (ubsan), see https://developers.redhat.com/blog/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan

VIRGINIA TECH

[2] I am not referring to looking for memory leaks with valgrind, which is a different topic altogether.

References

[1] Chris Lattner.
What every c programmer should know about undefined behavior.
https://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

[2] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama.
Towards optimization-safe systems: Analyzing the impact of undefined behavior.
In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 260–275, New York, NY, USA, 2013. Association for Computing Machinery.

[3] Victor Yodaiken.
How ISO C became unusable for operating systems development.
In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, PLOS '21, page 84–90. Association for Computing Machinery, 2021.

VIRGINIA TECH