Undefined Behaviour in C

— Report —

Field of work: Scientific Computing
Field: Computer Science
Faculty for Mathematics, Computer Science and Natural Sciences
University of Hamburg

Presented by:	Dennis Sobczak
E-mail-address:	1sobczak@informatik.uni-hamburg.de
Registration number:	6325177
Course of study:	Computer Science

First surveyor:	
Second surveyor:	
Supervisor:	Konstantinos Chasapis

Hamburg, 31.03.2014

Contents

1 Preface

The C programming language was designed to be an extremely efficient low-level programming language. Because of this there are things possible which may cause unexpected behaviour of the program. Therefore C is called 'unsafe' in opposite to Java for instance, where the compiler restricts code evoking such behaviours. The kind of behaviour which will be described in the following is referred to as 'undefined behaviour'.

2 What is meant by 'undefined behaviour' in C?

Undefined behaviour is some kind of a 'dark side' of C. In general every C code which contains operations against the C norms and standards may evoke this undefined behaviour. So breaking these rules might end up in a mess if not recognized before. Even if the programmer is aware of such content in his own code he or she should avoid that and think about another possibility of implementing this current routine all the more if the person is dealing with safety critical code. There are many different possibilities to make the program result in an undefined behaviour and with this there exist many classes one can categorize. So in the end there will be at least bugs in the program which in turn may end in a system crash on certain CPUs or do other unexpected things. There are lot of prominent examples for that kind of bugs in the later chapters.

2.1 Norms and Standards

C has been developed in the early 1970's by computer scientist Dennis Ritchie at Bell Labs. It has spread very fast what meant that constant modifications and expansions were made and many versions of C have been created. These versions were not supported completely by every C compiler. Therefore Ritchie and co-author Kernighan had written down norms and standards, rules and restrictons, to keep faultless and reliable code. This should allow to run the same code on different architectures with any C compiler without any trouble with undefined behaviour. Nevertheless it is possible to evoke undefined behaviour - rules can be broken.

2.2 Why is undefined behaviour possible?

As mentioned in the preface before, C is an extremely efficient low-level programming language. Because of this it is not as safe as other programming languages. This means that also forbidden things are possible, which are restricted in any other programming language. Nevertheless using code which causes undefined behaviour might enable certain optimizations the compiler can benefit from. Optimizations can be code optimizations, compilation time optimizations, performance optimizations of the system and the applications and storage usage optimizations. How to reach that kind of optimizations will be shown in the next chapter.

3 How does the compiler benefit?

There are two commonly used compilers, the GCC (GNU Compiler Collection) and LLVM (Low-Level-Virtual-Machine) which is in focus from now on. The LLVM was developed by the LLVM Developer Group. It was written in C++. The frontend is Clang. In the following there are a few code examples which display some of the undefined behaviour classes.

3.1 Division by zero

Division by zero is according to the standard undefined. Therefore one should check his code if there are passages where division by zero is taking place.

Listing 3.1: Division by zero

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int divide_by_zero(int x,int y);
5
6  int main(){
7    int a,b;
8    a=1;
9    b=0;
10   printf("Result is: %d\n",divide_by_zero(a,b));
11   return 0;
12 }
13
14 int divide_by_zero(int x,int y){
15     int result;
16     result=x/y;
17     return(result);
18 }
```

If this code is compiled with no optimizations enabled, there will not occur any warning message during compilation time. But if the program is run afterwards it will result in a 'Floating point exception'.

3.2 Indexing arrays out of bounds

Indexing an array beyond its bounds is a commonly known source of error. If this happens in the code, dependend on which target architecture it is compiled, it can occur that the program will en up in a crash, another program is started or even more worse, the harddrive may be formatted. It also appears to be an advantage for hackers to manipulate the program, to large data packets are stored in a far too small reserved buffer which causes an overwrite of other store cells. This kind of attack is known as bufferoverflow.

Listing 3.2: Bufferoverflow

```
1  void input_line()
2  {
3      char line[1000];
4      if (gets(line))     // gets receives a pointer to the
5          parse_line(line); //array, no size information
6  }
```

Before the subroutine is executed the return address is written to the stack. After that the local variables are written to the stack, the stack itself growing downwards. If fields or strings are used, they are written upwards in the stack. The attacker may provoke this gap in security and get back to the return address which then can be modified.

To prevent this the array should be range checked before accessing the current array element and in such an error case an exception should be thrown.

3.3 Casting types

A type cast can reduce the size of used storage for example if an integer (2 or 4 bytes) is casted to a char (1 byte):

Listing 3.3: Casting an integer into a char

```
1
2  int main()
3  {
4     int *num;
5     char *charPtr;
6     charPtr = (char*) num;
7     *charPtr = (char*) 0;
8     *charPtr = (char*) 1;
9     return 0;
10 }
```

Another advantage is that Type-Based Alias Analysis (uses the types of variables to determine what things might alias what) gets enabled. The disadvantage is similar to

the example above, so is the solution again the same: Check the size of the variable before accessing it.

3.4 Using uninitialized variables

In C variables which are not initialized directly in the code will not be initialized during compilation time with a default value as known from other programming languages. This means that there are no zero-initializations what in effect makes the program run faster. A disadvantage is that this can cause overhead for stack arrays. Todays compilers have special sections to omit the zero-initialization by marking the variable with the right keyword according to the programming langauge used.

3.5 Signed integer overflow

Listing 3.4: Signed integer overflow

```
1  #include <limits.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7    int number = INT_MAX;
8    result = number + 1;
9    printf("Result is: %d\n",result);
10   return 0;
11 }
```

Compiled with clang (no optimizations turned on) the result is '-2147483648',

`INT_MIN`

. This result is not guaranteed. On some architectures the result is undefined, what means that it is something oter than

`INT_MIN`

, no wraparound is done. If such code is used for example in a for loop this may end up in an infinite loop.

3.6 Oversized shift amounts

Shifting values by an amount greater or equal to the number of bits in the number can evoke undefined behaviour. Depending on the used platform that kind of shift may be a

shift by zero or in the worst case format your harddrive. There are two possibilities to prevent this. The first one requires knowledge of the bitwidth of the current type so it can be checked before shifting by a certain value. The second possibility is to set the variables to zero with a left shift (lsl).

3.7 Dereferencing a NULL Pointer

Dereferencing a NULL pointer enables certain optimizations like scalar optimizations exposed by macro exapnasion and inlining. But there is also the danger of a system crash doing that.

Listing 3.5: Dereferencing a NULL Pointer

```
1 int* ptr = NULL;
2 int& ref = *ptr;
3 int* ptr2 = &ref;
```

3.8 Dereferencing a Dangling Pointer

Dereferencing dangling pointers can enable similar optimizations. The danger is that memory can be overwritten.

Listing 3.6: Dereferencing a Dangling Pointer

```
1 int* foo(){
2     int y;      //decalring y as an integer
3     return &y; //returning the address of y
4 }
5
6 int main(){
7    int* pY = foo();  //declaring a pointer pY, which
8                      //points to the address of y
9 }
```

4 Dangers

4.1 Interacting compiler optimizations

The interacting compiler optimization can be either helpful or it can cause sideeffects in the program, which then can be exploited by attackers. Optimizers are different from compiler to compiler. Optimization is run at various stages and in different order. So it did in the prominent example below:

Listing 4.1: Linuxkernel: Checking for the NULL Pointer

```
1 void contains_null_check(int *P){
2     int dead = *P;
3     if(P == 0)
4     return;
5     *P = 4;
6 }
```

This code fragment caused an exploitable bug in the Linuxkernel. There are two optimization sequences which become problematic if run in a certain order. It is about the 'Dead Code Elimination' (DCE) and the 'Redundant Null Check Elimination' (RNCE). The first function, as the name is indicating, checks the code for possible dead code (code which does not have any influence on the result) eliminates it. The second one checks the code for redundant code fragments and eliminates these. So refered to the bug in the Linuxkernel above: If the DCE is run before the RNCE the second line is deleted. If the RNCE is run afterwards the following null check (line 3) is not redundant and consequently kept. However, if the processing is different, i. e. if the compiler is structured differently, the RNCE could be run before the DCE what means the if condition above is always false, because 'P' was dereferenced and can not be null. The DCE follows and all is left is line 4, *P = 4;. To prevent this the volatile keyword could precede the declaration in the second line, which will signalize the compiler that the value will change during runtime.

4.2 Security

It is not a good idead to use undefined behaviour in security-critical code if it is the intention. Undefined behaviour can make a system vulnerable: If an attacker figures out this vulnerability he or she can exploit it. Another contra are totally unexpected system crashes that may appear during utilization.

```
1  void process_something(int size){
2       if(size > size + 1)
3       abort();
4       ...
5
6       char *string = malloc(size + 1);
7       read(fd, string, size);
8       string[size] = 0;
9       do_something(string);
10      free(string);
11 }
```

In the code above could result a similar bug as in the example before. malloc is checking whether an integer overflow occurs. Through optimizations the if condition in line 2 as well as the abort() in the following line are eliminate, just the lines from line 6 on are kept. This makes the program again vulnerable to exploits. Instead of checking the condition 'size > size + 1' the condition should be turned to

`'size == INT_MAX'`

, which will rectify this fault.

4.3 Changing compiler without adapting the code

If the code is used on a different compiler, it should be adapted before compiling it. Another compiler can have different features than the previous one. The differences mainly lie in optimizations techniques and diagnostics.

5 What one should be aware of?

The compiler does not always benefit from optimizations. As seen before in the code examples the compilers optimization strategies can cross plans if it is eliminating important code fragment, when it is seen as dead code or redundancy. Furthermore there are no specially-tailored warning messages that could mark code that will evoke undefined behaviour because there are far too many cases that would need to have a special warning messages.

6 Clang options to avoid undefined behaviour

There are a few options for clang which enable some kind of detection of passages in the code which can damage the program:

- -fcatch-undefined-behaviour (-ftrapv)
  - it is experimental
  - it detects this code (and traps it)
  - makes runtime checks (checking for shift-out-of-range etc.)
  - because of the the applications runtime is slowed down
- -fwrapv
  - trapping signed integer overflow at runtime

7 Code Analyzers

There are static and dynamic analyzers which can help to detect undefined behaviour:

- Clang Static Analyzer:
  - CSA makes a deep analyzis
  - includes checking for undefined behaviour (i. e. finds null pointer dereferences)
  - does not generate information at runtime
  - is not integrated into normal workflows
- Valgrind:
  - Valgrind is a dynamic analyzer which generates information at runtime
  - finds uninitialized variables and other memory bugs
  - quiet slow
  - it can not find bugs, which the optimizer removes, only bugs that still exist in the machine code
  - is not aware of the source language what makes it more difficult to notice shift-out-of-range or signed integer overflow

8 Recommendations

Todays compilers optimize a lot automatically so not all of the presented examples will have these side-effects anymore when tried out. After all we have seen you should keep a few things in mind: Inform yourself about the compiler you want to use, how its optimization strategies work and how warning messages are displayed which you should also turn on. A well documented code is always a great help especially when you document preconditions and postconditions (assertions). And of course debug and test your program again and again.

9 References

http://de.wikipedia.org/wiki/Puffer%C3%BCberlauf
http://www.drdobbs.com/cpp/type-based-alias-analysis/184404273
http://stackoverflow.com/questions/14686030/is-it-possible-to-instruct-c-to-not-ze
http://stackoverflow.com/questions/8205858/clang-vs-gcc-for-my-linux-development-p
http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html
http://blog.regehr.org/archives/213
http://en.wikipedia.org/wiki/Undefined_behavior
http://stackoverflow.com/questions/2727834/c-standard-dereferencing-null-pointer-t