Security in Software
Posix, sanitizers, typical system programming errors, fuzzing

Ole Christian Eidheim

IDI, NTNU

September 28, 2024

Overview

- **POSIX and compilation systems**
- Sanitizers
- Integer overflow
- Range types
- Contracts
- Buffer overflow
- Buffer overread
- Fuzzing
- Exercise 5

Portable Operating System Interface (POSIX)

- IEEE standard for operating systems compatibility
  - most major OSs apart from Windows support this standard
  - POSIX may be too extensive for microkernels:
    - Arduino does not support the POSIX standard
    - Magenta kernel of Google Fuchsia has POSIX compatibility features
  - POSIX contains for example standardized:
    - processes
    - threads
    - (a)synchronous I/O
    - command line interpretation (for instance: echo “hello” > file.txt)
    - C libraries
    - Segmentation / Memory Violations
    - command line tools (e.g. grep, kill, ls, sort, tail, vi)
  - Linux is largely POSIX compliant
    - Linux Standard Base: contains extensions that are not part of POSIX
  - MacOS is POSIX certified

POSIX based compiler systems - collection of compilers, libraries and tools

- GNU Compiler Collection (GCC) (1987)
  - gcc/g++
    - The most common C/C++ compiler
- Low Level Virtual Machine (LLVM) (2003)
  - clang/clang++
    - Standard in FreeBSD and MacOS
    - Windows supported, Chrome for Windows kompileres nå med clang++ (2018)
  - Used by the Rust compiler
  - Various support tools:
    - LLDB - less resource-intensive debugger
    - Clang Static Analyzer - more thorough static analysis
    - Various programs to format/tidy/analyze source code

Overview

- POSIX and compilation systems
- **Sanitizers**
- Integer overflow
- Range types
- Contracts
- Buffer overflow
- Buffer overread
- Fuzzing
- Exercise 5

Sanitizers - Runtime checks

- Address sanitizer - keeps track of used memory when running the program, and interrupts the program with an error message when an invalid memory operation is performed
  - For example buffer overflow/overread
  - Valgrind - more comprehensive program for finding such errors
  - Eksempler: buffer-overrun - demonstrates use of address sanitizer and Valgrind
- Thread sanitizer - detects *data races*
  - For example, when a variable is altered without using a mutex in two different threads
- Undefined behavior sanitizer - finds for example
  - Signed/unsigned integer overflow
  - Conversion overflow
  - Division by 0
- Both GCC and LLVM contain sanitizers
- Sanitizers are activated solely during testing due to increased resource usage

Overview

- POSIX and compilation systems
- Sanitizers
- **Integer overflow**
- Range types
- Contracts
- Buffer overflow
- Buffer overread
- Fuzzing
- Exercise 5

Integer overflow
- C/C++: prioritize more effective machine code

- Integer overflow is one of many undefined behaviors that must be avoided in critical software

```
#include <iostream>

int main() {
  int num = 2147483647;
  num += 1;
  std::cout << num << std::endl; // What is the output here?
}
```

- Some simple integer overflows can be detected at compile time
- Can be detected through Undefined behaviour sanitizer at runtime

Integer overflow
- Rust: does not accept undefined behaviour

```
fn main() {
  let mut num: i32 = 2147483647;
  num += 1; // Program terminates
  println!("{}", num);
}
```

- Some simple integer overflows can be detected at compile time
- At runtime, the program terminates if an integer overflow occurs

Integer overflow
- Arbitrary-precision arithmetic

- Languages like Python avoid the problem through number types that can contain numbers of any size (memory is the only limitation)
- Most programming languages have library support for arbitrary-precision arithmetic.
    - In C++: Boost.Multiprecision
- Resource demanding

Overview

- POSIX and compilation systems
- Sanitizers
- Integer overflow
- **Range types**
- Contracts
- Buffer overflow
- Buffer overread
- Fuzzing
- Exercise 5

Range types in Ada

- Ada: widely used programming language for critical software
  - Used for example in aircraft and flight control software, satellites and banking systems

Range type example in Ada:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type My_Int is range 1 .. 20;
   A : My_Int := 12;
   B : My_Int := 15;

   M : My_Int := (A + B) / 2; -- No overflow here

   N : My_Int := A + B;       -- Will cause compiler warning and
                              -- exception at runtime
begin
   Put_Line ("Hello World!");
end Main;
```

Range types in C++

- bounded::integer
  - "provides more safety than Ada range types and more efficiency than the C / C++ built-in integer types."

Example:

```
bounded::integer<0, 10> num(5);
num+=10; // Compile time error or runtime exception
```

Overview

- POSIX and compilation systems
- Sanitizers
- Integer overflow
- Range types
- **Contracts**
- Buffer overflow
- Buffer overread
- Fuzzing
- Exercise 5

Contracts in Ada

- In Ada, you can define contracts for functions in the form of *pre-* and *post-conditions.*

Example:

```
generic
   type Item is private;
package Stacks is
   type Stack is private;
   function Is_Empty(S: Stack) return Boolean;
   function Is_Full(S: Stack) return Boolean;
   procedure Push(S: in out Stack; X: in Item)
      with
         Pre => not Is_Full(S),
         Post => not Is_Empty(S);
   procedure Pop(S: in out Stack; X: out Item)
      with
         Pre => not Is_Empty(S),
         Post => not Is_Full(S);
private
      ...
end Stacks;
```

Contracts in C++

- Might arrive in the 2026 standard
  - Can be activated during testing, and deactivated in production (for less resource use)
  - Can already be tried out in Compiler Explorer

Example:

```
int f(int n)
  [[pre: n >= 0]]
  [[post r: n == r]]
{
  return n;
}

int main() {
  f(-2); // Contract violation
}
```

Overview

- POSIX and compilation systems
- Sanitizers
- Integer overflow
- Range types
- Contracts
- **Buffer overflow**
- Buffer overread
- Fuzzing
- Exercise 5

Buffer overflow

```
char           A[8] = {};
unsigned short B = 1979;
```

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

```
strcpy(A, "excessive");
```

variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

Buffer overflow
- editing data, the stack, example 1

```
#include <iostream>

using namespace std;

int main() {
  char chr = 'a';
  char table[3] = {1, 2, 3};

  cout << "table starts at memory address "
       << &table << endl;
  cout << "chr starts at memory address "
       << (void *)&chr << endl;

  table[3] = 'b';

  cout << chr << endl;
}
```

Output e.g:
table starts at memory address 0x7fff51ed6b98
chr starts at memory address 0x7fff51ed6b9b
b

- Protection of objects in the stack
  - Use high-level programming if possible
    - Foreach loops or functional algorithms
  - Use support tools such as:
    - AddressSanitizer
    - Valgrind
  - The compiler may issue warnings/error messages

Buffer overflow
- editing data, the stack, example 2

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

char input[20];
int success;

int main() {
  success = 0;

  printf("Password: ");                        // Print to standard output
  fflush(stdout);                              // Flush standard output
  scanf("%s", input);                          // Read from standard input,
                                               // and store the data in the variable input
  if (strcmp(input, "PassWord213") == 0) // If input is equal to: PassWord213
    success = 1;

  if (success)
    printf("Logged in\n");
  else
    printf("Wrong password\n");
}
```

Buffer overflow
- editing data, the heap

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<char> table = {1, 2, 3};
  char *chr = new char('a');

  cout << "table.data() starts at memory address "
       << (void *)table.data() << endl;
  cout << "chr starts at memory address "
       << (void *)chr << endl;

  table[16] = 'b';

  cout << *chr << endl;
}
```

Output e.g.:
table.data() starts at memory address 0x7f9929500000
chr starts at memory address 0x7f9929500010
b

- Protection of data in the heap
  - Use high-level programming if possible
    - Foreach loops or functional algorithms
  - Use support tools such as:
    - AddressSanitizer
    - Valgrind
  - Range check with exception possible
    - e.g. `table.at(16)`

Buffer overflow
- editing function pointers

IKKE KJØR DENNE KILDEKODEN

```
#include <iostream>

using namespace std;

void func1() {
  cout << "Output from func1" << endl;
}

void func2() {
  cout << "Output from func2" << endl;
}

int main() {
  cout << "The function func1 starts at memory address " << (void *)&func1 << endl;
  cout << "The function func2 starts at memory address " << (void *)&func2 << endl;

  auto f = &func1;
  char table[3] = {1, 2, 3};

  cout << "The pointer f starts at memory address " << (void *)&f << endl;
  cout << "The table starts at memory address " << (void *)&table << endl;

  f();

  table[3] = (size_t)&func1 + 64;

  f();
}
```

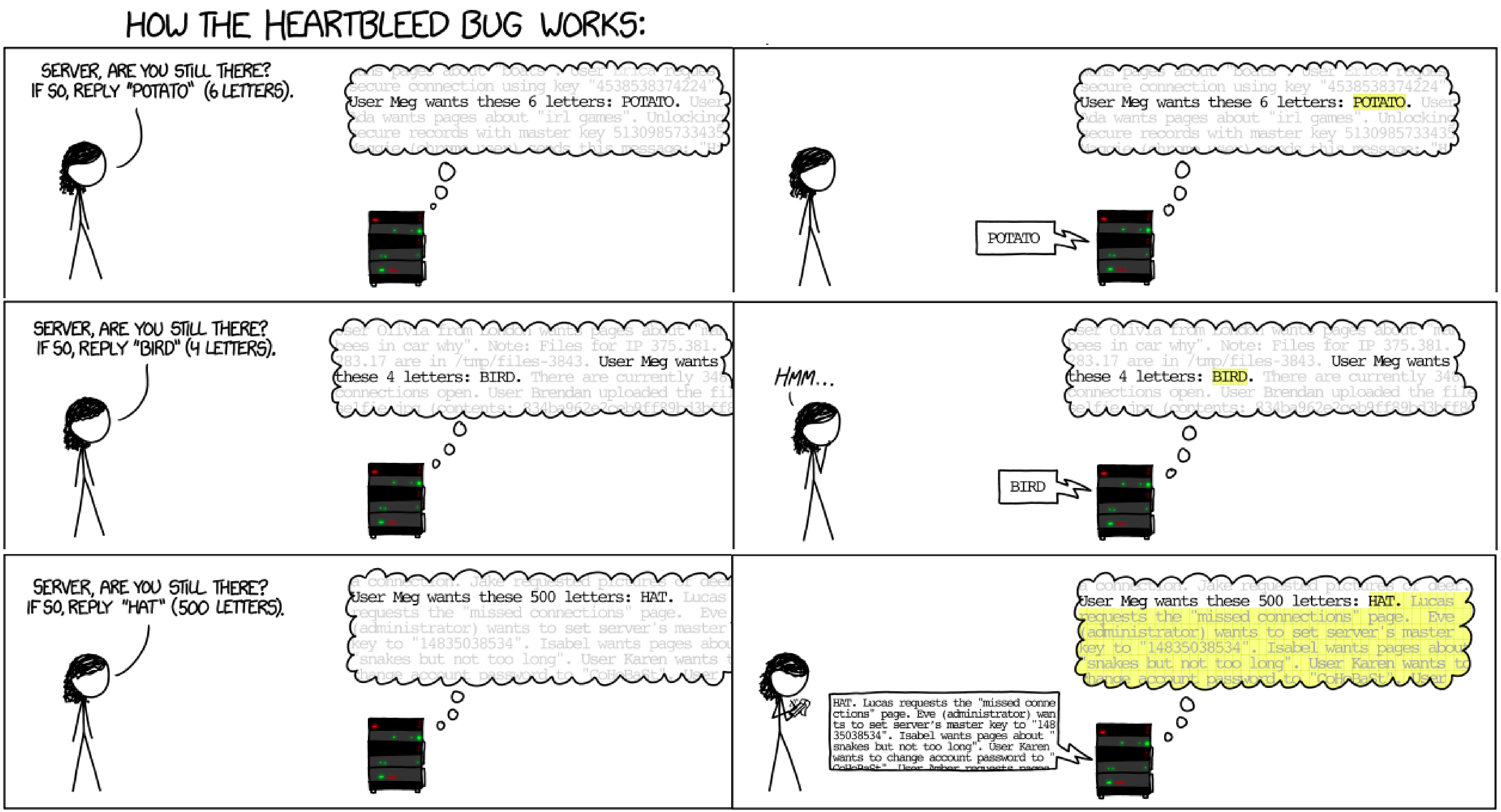
Output e.g.:
The function func1 starts at memory address 0x10076ff90
The function func2 starts at memory address 0x10076ffd0
The pointer f starts at memory address 0x7fff5f490b80
The table starts at memory address 0x7fff5f490b7d
Output from func1
Output from func2

Overview

- POSIX and compilation systems
- Sanitizers
- Integer overflow
- Range types
- Contracts
- Buffer overflow
- **Buffer overread**
- Fuzzing
- Exercise 5

Buffer overread

- Heartbleed (OpenSSL vulnerability for versions <1.0.1g)



Buffer overread - Heartbleed-like example

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string secret_data="secret key!";

  string str;
  int length;
  cin >> str >> length;

  for(int c = 0; c < length; c++)
    cout << str[c];

  cout << endl;
}
```

Input:

hello 40

Output e.g.:

hello????????????????? secret key!

Overview

- POSIX and compilation systems
- Sanitizers
- Integer overflow
- Range types
- Contracts
- Buffer overflow
- Buffer overread
- **Fuzzing**
- Exercise 5

Fuzzing

- Automatic testing of functions through generated input parameters
  - Popular tool: libFuzzer
    - Have been used to find bugs in widely used software
    - Works together with sanitizers
    - Advanced functionality: "... the fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize the code coverage."
- See fuzzing-example

Overview

- POSIX and compilation systems
- Sanitizers
- Integer overflow
- Range types
- Contracts
- Buffer overflow
- Buffer overread
- Fuzzing
- **Exercise 5**

Exercise 5

- Perform *fuzzing* with *address sanitizer* on the C function you created in exercise 4b)
  - Fix bugs you find through fuzzing, or introduce bugs that are discovered through fuzzing
  - **Voluntary**: Set up a CI (Continuous Integration) solution that performs *fuzzing* with *address sanitizer*
    - See instructions for running fuzzing in a terminal (note that you can limit the number of seconds libFuzzer will run with the `-max_total_time` argument)
    - Create tests, and run the tests as well in CI (fuzzing-example contains an example test)