Hacking in C

Reflections on using C(++)

Root Cause Analysis
Abstractions  Complexity  Assumptions  Trust
“There are only two kinds of programming languages: the ones people complain about and the ones nobody uses.”

Bjarne Stroustrup, the creator of C++
What we have seen in this course

Some complexities that hide under the hood of C:

- **representation of data types**, eg
  - `long` as big/little endian sequences of bytes
  - string, or `char*`, as `char` sequence terminated by the null character `0`

- **allocation of data** on stack (by default) and heap (with `malloc`)

- **execution of code** - esp. function calls - using the stack
  - with return addresses and frame pointers

with some consequences

- **unexpected interpretation of data**
  - implicit casts between data types, `p+1` for pointers `p`, `%n` `%s` `%p` in format strings, undefined behaviour

- **unintended manipulation of data**
  - array index outside bounds, pointer arithmetic,
    accessing uninitialized or de-allocated memory, memory leaks

which allows attacks
The good news

C is a small language that is close to the hardware

• you can produce highly efficient code
• compiled code runs on raw hardware with minimal infrastructure

Therefore C is typically the programming language of choice

• for highly efficient code
• for embedded systems (which have limited capabilities)
• for system software (operating systems, device drivers,...)
The somewhat bad news

The precise semantics of C programs depends on underlying hardware, eg
- sizes of data types differ per architecture
- casting between types will reveal endianness of the platform
- ...

The *precise* semantics of a C program can only be determined by looking at
1) the compiled code
2) the underlying hardware

For *efficiency* it is unavoidable you have to know the underlying hardware, but for the *semantics* you’d wish to avoid this.

This also hampers *portability* of code
The really bad news

Writing *secure* C(++) code is hard, because the built-in notorious sources of security vulnerabilities

- *buffer overruns*, and *absence of array bound checks*

- *dynamic memory*, managed by the programmer with `malloc` and `free` and using complex pointers

More generally: *undefined behavior* caused by this

And format string attacks, though these should be easy to fix..
The good vs the bad news

“C is a terse and unforgiving abstraction of silicon”
Undefined behaviour

Defined in the FAQ as

Anything at all can happen; the Standard imposes no requirements.
The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.
Undefined causing problems [Example from Linux kernel]

```c
1. unsigned int tun_chr_poll(struct file *file,
2.                  poll_table *wait)
3. {
4.   struct tun_file *tfile = file->private_data;
5.   struct tun_struct *tun = __tun_get(tfile);
6.   struct sock *sk = tun->sk;  // shorthand for (*tun).sk
7.   if (!tun) return POLLERR;  // check if tun is non-null
8.   ...
   .. }
```

Line 7 checks if `tun` is NULL, and if so, returns error

But in line 6 `tun` is already de-referenced...

Compiler is now allowed to assume that `tun` is non-null in line 7
• because if `tun` is null, then line 6 is undefined behaviour, and any code the compiler produces is ok
  gcc will actually remove line 7 as compiler optimisation.

This caused a security flaw in Linux kernel when compiled with gcc.
Undefined causing problems [Example from Linux kernel]

The code below is careful to check for negative lengths and potential integer overflows

1. int vsnprintf(char *buf, size_t size, ...) {
2.     char *end;
3.     if (size < 0) return 0; // reject negative length
4.     end  = buf + size;
5.     /* Make sure end is always >= buf */
6.     if (end < buf ) {...}
7.     ...
8. }

Programmer *assumes* that if *buf+size* overflows, it will become negative. However, *buf+size* overflowing is undefined behaviour, and *all bets are off*. Compiler may remove line 6; it knows that *size* is non-negative and may assume that *buf+size >= buf* for any addition with a non-negative number

- because if the addition overflows, this is undefined behaviour, and any code it produces is ok
C(++) vs safer languages

You can write insecure code in any programming language.

Still, some programming languages offer more in-built protection than others, eg against

- buffer overruns, by checking array bounds
- problems with dynamic memory, eg by garbage collection
- missing initialisation, by offering default initialisation
- suspicious type casts, by disallowing these
- integer overflows, by raising exceptions when these occur

C(++) programmer is like trapeze artist without safety net
Consequences of the bad news

Many products should carry a government health warning

*Warning: this product contains C(++) code and therefore, unless the programmers were experts and never made a mistake, is likely to contain buffer overflow vulnerabilities.*

As a C(++) programmer, you have to become an expert at avoiding classic security flaws in these languages

- incl. using all the defenses discussed last week
C(++) secure coding standards & guidelines

Fortunately, there is now good info available, eg

C(++) Secure Coding Standards by CERT
https://www.securecoding.cert.org

NB

• If you are going to write C(++) code, you have to read such documents!

• If you work for a company that produces C(++) code, you’ll have to make sure that all programmers read them too!
Dangerous C system calls

Extreme risk
- gets

High risk
- strcpy
- strcat
- sprintf
- scanf
- sscanf
- fscanf
- vfscanf
- vsscanf
- streadd
- strcpy
- strtrns
- realpath
- syslog
- getenv
- getopt
- getopt_long
- getpass

Moderate risk
- getchar
- fgetc
- getc
- read
- bcopy

Low risk
- fgets
- memcpy
- snprintf
- strccpy
- strccadd
- strncpy
- strncat
- vsnprintf
C(++) secure coding standards & guidelines

More general coding guidelines, which also cover other (OS-related) aspects besides generic C(++) issues:

• **Secure programming HOWTO** by Dan Wheeler
  chapter 6 on buffer overflows
  Linux/UNIX oriented

• **Writing Secure Code** by Howard & Leblanc
  chapter 5 on buffer overflows
  Microsoft-oriented
Some root cause analysis
Recurring theme: functionality vs security

There is often a tension between functionality and security.

People *always* choose for functionality over security:

Classic example:

- efficiency of the language (not checking array bounds)
  vs
- security of the language (checking array bounds)
functionality vs security
Recurring theme: 

Mixing control data (namely return addresses & frame pointers) and untrusted user data (which may overrun buffers) next to each other on the stack was not a great idea.

Remember the root cause of phone phreaking!

Note that format string attacks also involve control data (or control characters) inside user input.
Recurring theme: complexity

- Who understands all implicit conversions between C data types?
- Who can understand whether a large C program leaks memory? Or accidentally accesses freed memory? (because there are too many/too few/the wrong `free` statements)
- Who understands all the sources of possibly undefined behaviour?
- Who understands the compiler optimalisations that are allowed in the presence of undefined behavior?
- Should a programmer have to know the entire C language specification to be able to write secure code?

**Complexity is a big enemy of security**

Abstractions are our main (only?) tool to combat – or at least control – complexity.
Controlling complexity: Abstractions

We want to deal with abstractions instead of complex underlying representations, eg

- a `long` instead of a big- or little-endian sequence of `sizeof(long)` bytes
- a `string "Hello"` instead of a null-terminated sequence of
- array indexing, eg. `a[12]`, instead of pointer arithmetic in `&a+12*sizeof(int)`
- a function call `f(12)` instead of
  - push 12 on the stack
  - push current address & frame pointer on stack
  - allocate local variables and jump to `f`
  - upon return, pop everything from the stack & continue at the return address you found on the stack
Abstractions

Ideally, abstractions should be rock solid, so they cannot be broken.

This means

- they do not rely on assumptions (on how they are used)
- the underlying representation does not matter, and is never revealed
  - even when users supply malicious input.

If that is the case we do not have to trust programs, or the programmers that write them, to use abstractions in the right way.
Implicit assumptions

Often abstractions rely on implicit assumptions, which can be broken. For example, assuming that:

- there is infinite stack memory, so function calls never fail
- there is infinite heap memory, so a `malloc` never fails, and `free`’s are not really needed
- `int`’s are mathematical integers
- strings fit into buffers
- the return address on the stack still points to the right place
- a `char` array has a null terminator
- a pointer is not null
- ....

Implicit invalid assumptions are important cause of security problems
Trust

Is trust good? Is trust bad?

ie. do we want as much trust as possible, or as little trust as possible?
Trust vs trustworthiness

vertrouwen vs betrouwbaarheid

Trust backed by trustworthiness is good. Trust without trustworthiness is bad. We want to minimise trust, because trust in something means that that something could damage you.

TCB (Trusted Computing Base) is the collection of software & hardware that we have to trust for a system to be secure. We want the TCB to be as small as possible. Of course, we also want our TCB to be as trustworthy as possible.
Trusted Computing Base (TCB)

• The Trusted Computing Base of a C(++) program includes all the code, incl. libraries
  – because any buffer overflow, flawed pointer arithmetic, ...
somewhere can effect the memory anywhere

• The Trusted Computing Base also includes
  – the compiler
  – the OS
  – the hardware
Reflection on trusting trust

Title of Ken Thompson’s Turing award acceptance speech, where he revealed a backdoor in UNIX and Trojan in C-compiler.

1. backdoor in login.c:
   
   if (name == "ken") {don't check password; log in as root}

2. code in C compiler to add backdoor when recompiling login.c

3. code in C compiler to add code (2 & 3!) when (re)compiling a compiler

Moral of the story: you may be trusting more than you expect!

Trust is transitive
Overview: recurring themes in insecurity

• complexity
  – incl. unexpected interpretation of special characters
    \n \s ; \ \ \ | .

• functionality vs security
• mixing user and control data
• abstractions that can be broken
• (misplaced) trust in these abstractions
• trust in general, and not realising what the TCB is