University of Glasgow

# Security Considerations

Advanced Systems Programming (M)

Lecture 8

Colin Perkins | https://csperkins.org/ | Copyright © 2019 | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

# Lecture Outline

- Memory Safety
- Parsing and LangSec
- Modern Type Systems and Security

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Safety

CC BY ND

# Memory Safety in Programming Languages

- A **memory safe** language is one that ensures that the only memory accessed is that owned by the program, and that such access is done in a manner consistent of the declared type of the data
  - The program can access memory through its global and local variables, or through explicit references to them
  - The program can access heap memory it allocated via an explicit reference to that memory
  - All accesses obey the type rules of the language
    - Array bounds are respected
    - Memory cannot be accessed after it's freed
    - References can only refer to objects of matching type
    - …

| Memory Safe | Memory Unsafe |
|---|---|
| Java, Scala, C#, Rust, Go, Python, Ruby, Tcl, FORTRAN, COBOL, Modula-2, Occam, Erlang, Ada, Pascal, Haskell, … | C, C++, Objective-C |

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation

```
double *d = malloc(sizeof(float));
```

A call to **`malloc()`** does not check if the amount of memory allocated corresponds to the size required to store the type

A memory safe language will require the size of the allocation to match the size of the allocated type at compile time

Operation ought to be “allocate enough memory for an object of type **`T`** and return a reference-to-**`T`**” and not “allocate *n* bytes of memory and return an untyped reference”

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation

```
char *buffer;

int rc = recv(fd, buffer, BUFLEN, 0);
if (rc < 0) {
    perror("Cannot receive");
} else {
    // Handle data
}
```

Passes a pointer to the **`buffer`** to the **`recv()`** function, but forgets to **`malloc()`** the memory – or assumes it'll be allocated in **`recv()`**

Memory safe languages require that all references be initialised and refer to valid data – good C compilers warn about this too

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**

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

int main() {
  char *x = malloc(14);
  sprintf(x, "Hello, world!");
  free(x);
  printf("%s\n", x);
}
```

Accesses memory that has been explicitly freed, and hence is no longer accessible

Automatic memory management eliminates this class of bug

```
use std::mem::drop; // free() equivalent

fn main() {
  let x = "Hello".to_string();
  drop(x);
  println!("{}", x);
}
```

```
error[E0382]: use of moved value: `x`
 --> test.rs:6:18
  |
5 |   drop(x);
  |        - value moved here
6 |   println!("{}", x);
  |                  ^ value used here after move
  |
  = note: move occurs because `x` has type `std::string::String`, which does not implement the `Copy` trait
```

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free

```
int *foo() {
  int n = 42;
  return &n;
}
```

Return reference to stack allocated memory that is used after the stack frame has been destroyed

Automatic memory management eliminates this class of bug

```
fn foo() -> &i32 {
  let n = 42;
  &n
}
```

```
% rustc test.rs
error[E0106]: missing lifetime specifier
 --> test.rs:1:13
  |
1 | fn foo() -> &i32 {
  |             ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no
  = help: value for it to be borrowed from
```

cc BY ND

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type

```
char *buffer = malloc(BUFLEN);

int rc = recv(fd, buffer, BUFLEN, 0);
if (rc < 0) {
    …
} else {
    struct pkt *p = (struct pkt *) buffer;
    if (p->version != 1) {
        …
    }
    …
}
```

Common in C code to see casts from **`char *`** buffers to a more specific type (e.g., to a **`struct`** representing a network packet format)

Efficient – no copies of memory – but unsafe since makes assumptions of **`struct`** layout in memory, size of block being cast

Memory safe language disallow arbitrary casts – write conversion functions instead; eliminates undefined behaviour

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values

```
// Send the requested file
while ((rlen = read(inf, buf, BUFLEN)) > 0) {
  if (send_response(fd, buf, strlen(buf)) == -1) {
    return -1;
  }
}
```

Strings in C are zero terminated, but **`read()`** does not add a terminator; buffer overflow results

Memory safe languages apply string bounds checks – runtime exception; safe failure, no undefined behaviour

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow

```
#define BUFSIZE 256
int main(int argc, char *argv[]) {
    char *buf;
    buf = malloc(sizeof(char) * BUFSIZE);
    strcpy(buf, argv[1]);
}
```

Memory safe languages apply bounds checks to heap allocated memory – runtime exception; safe failure, no undefined behaviour

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow
  - Array bounds overflow

```
int main(int argc, char *argv[]) {
    char buf[256];
    strcpy(buf, argv[1]);
}
```

Memory safe languages apply array bounds checks – runtime exception; safe failure, no undefined behaviour

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow
  - Array bounds overflow
  - Arbitrary pointer arithmetic

```
int buf[INTBUFSIZE];
int *buf_ptr = buf;

while (havedata() && (buf_ptr < (buf + sizeof(buf)))) {
  *buf_ptr++ = parseint(getdata());
}
```

**`sizeof()`** returns size in bytes, arithmetic on **`int *`** pointers is on pointer sized values; bounds check is too lax

Memory safe languages disallow arbitrary pointer arithmetic

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow
  - Array bounds overflow
  - Arbitrary pointer arithmetic
  - Use of uninitialised memory

Memory allocated with **`malloc()`** has undefined contents

Memory safe languages require memory to be initialised, or mandate that the runtime fills with known value

```
static char *
read_headers(int fd)
{
  char     buf[BUFLEN];
  char    *headers   = malloc(1);
  size_t   headerLen = 0;
  ssize_t  rlen;

  while (strstr(headers, "\r\n\r\n") == NULL) {
    rlen = recv(fd, buf, BUFLEN, 0);
    if (rlen ==  0) {
      // Connection closed by client
      free(headers);
      return NULL;
    } else if (rlen < 0)  {
      free(headers);
      perror("Cannot read HTTP request");
      return NULL;
    } else {
      headerLen += (size_t) rlen;
      headers = realloc(headers, headerLen + 1);
      strncat(headers, buf, (size_t) rlen);
    }
  }
  return headers;
}
```

CC BY ND

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow
  - Array bounds overflow
  - Arbitrary pointer arithmetic
  - Use of uninitialised memory
  - Use of memory via dangling pointer
  - Use of memory via **`null`** pointer

The **`lookup()`** call may fail, returning **`null`** pointer

Memory safe languages either: fail safely with an exception; or use **`Option<>`** types to enforce that the null pointer check is done

```
struct user *u = lookup(db, key);
printf("%s\n", u->name);
```

CC BY ND

# Memory Unsafe Languages

- What types of memory unsafe behaviour can occur in C and C++?
  - Type unsafe allocation
  - Use before allocation
  - Use after explicit **`free()`**
  - Use after implicit free
  - Use of memory as the wrong type
  - Use of string functions on non-string values
  - Heap allocation overflow
  - Array bounds overflow
  - Arbitrary pointer arithmetic
  - Use of uninitialised memory
  - Use of memory via dangling pointer
  - Use of memory via **`null`** pointer
  - …? https://cwe.mitre.org/data/definitions/658.html

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Impact of Memory Unsafe Languages

- Lack of memory safety breaks the machine abstraction
  - With luck, program crashes – segmentation violation
  - If unlucky, memory unsafe behaviour corrupts other data owned by program
    - Undefined behaviour occurs
    - Cannot predict without knowing precise layout of program in memory
    - Difficult to debug
    - **Potential security risk** – corrupt program state to force arbitrary code execution

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Impact of Memory Unsafe Languages – Security (1/2)

**Vulnerabilities By Type**

| Year | # of Vulnerabilities | DoS | Code Execution | Overflow | Memory Corruption | Sql Injection | XSS | Directory Traversal | Http Response Splitting | Bypass something | Gain Information | Gain Privileges | CSRF | File Inclusion | # of exploits |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1999 | 894 | 177 | 112 | 172 | | | 2 | 7 | | 25 | 16 | 103 | | | 2 |
| 2000 | 1020 | 257 | 208 | 206 | | 2 | 4 | 20 | | 48 | 19 | 139 | | | |
| 2001 | 1677 | 403 | 403 | 297 | | 7 | 34 | 123 | | 83 | 36 | 220 | | 2 | 2 |
| 2002 | 2156 | 498 | 553 | 435 | 2 | 41 | 200 | 103 | | 127 | 74 | 199 | 2 | 14 | 1 |
| 2003 | 1527 | 381 | 477 | 371 | 2 | 49 | 129 | | | | | | | | |
| 2004 | 2451 | 580 | 614 | 410 | 3 | 148 | 291 | | | | | | | | |
| 2005 | 4935 | 838 | 1627 | 657 | 21 | 604 | 786 | | | | | | | | |
| 2006 | 6610 | 893 | 2719 | 663 | 91 | 967 | 1302 | | | | | | | | |
| 2007 | 6520 | 1101 | 2601 | 954 | 95 | 706 | 884 | | | | | | | | |
| 2008 | 5632 | 894 | 2310 | 699 | 128 | 1101 | 807 | | | | | | | | |
| 2009 | 5736 | 1035 | 2185 | 700 | 188 | 963 | 851 | | | | | | | | |
| 2010 | 4652 | 1102 | 1714 | 680 | 342 | 520 | 605 | 275 | 8 | 234 | 282 | 238 | 86 | 73 | 1493 |
| 2011 | 4155 | 1221 | 1334 | 770 | 351 | 294 | 467 | 108 | 7 | 197 | 409 | 206 | 58 | 17 | 557 |
| 2012 | 5297 | 1425 | 1459 | 843 | 423 | 243 | 758 | 122 | 13 | 343 | 389 | 250 | 166 | 14 | 624 |
| 2013 | 5191 | 1455 | 1186 | 859 | 366 | 156 | 650 | 110 | 7 | 352 | 511 | 274 | 123 | 1 | 205 |
| 2014 | 7946 | 1598 | 1574 | 850 | 420 | 305 | 1105 | 204 | 12 | 457 | 2104 | 239 | 264 | 2 | 401 |
| 2015 | 6484 | 1791 | 1826 | 1079 | 749 | 218 | 778 | 150 | 12 | 577 | 748 | 367 | 248 | 5 | 127 |
| 2016 | 6447 | 2028 | 1494 | 1325 | 717 | 94 | 497 | 99 | 15 | 444 | 843 | 600 | 87 | 7 | 1 |
| 2017 | 14714 | 3154 | 3004 | 2805 | 745 | 503 | 1516 | 274 | 11 | 629 | 1706 | 459 | 327 | 18 | 6 |
| 2018 | 16555 | 1852 | 3035 | 2451 | 400 | 516 | 2001 | 509 | 11 | 709 | 1374 | 247 | 461 | 31 | 4 |
| 2019 | 4 | 2 | | | | 1 | | | | | | | | | |
| Total | 110603 | 22685 | 30435 | 17226 | 5043 | 7438 | 13667 | 3823 | 162 | 5880 | 10104 | 4877 | 2123 | 2195 | 4333 |
| % Of All | | 20.5 | 27.5 | 15.6 | 4.6 | 6.7 | 12.4 | 3.5 | 0.1 | 5.3 | 9.1 | 4.4 | 1.9 | 2.0 | |

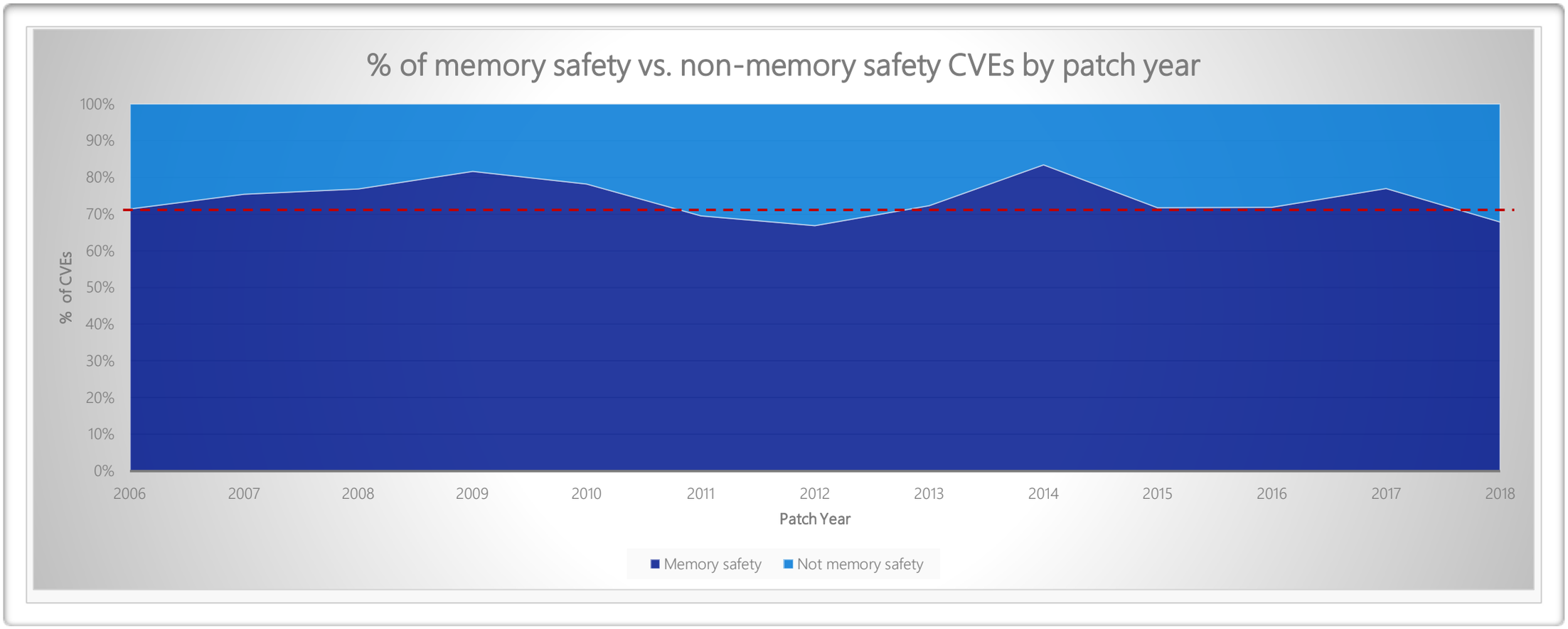
Source: https://www.cvedetails.com/vulnerabilities-by-types.php

Half of *all* reported security vulnerabilities are memory safety violations that should be caught by a modern type system – buffer overflows, use-after-free, memory corruption, treating data as executable code

Use of type based modelling of the problem domain can help address others – by more rigorous checking of assumptions

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Impact of Memory Unsafe Languages – Security (2/2)




Source: Matt Miller, Microsoft, presentation at BlueHat IL conference, February 2019
https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL

- ~70% of Microsoft security updates fix bugs relating to unsafe memory usage
  - Interestingly, this has not significantly changed in >10 years
  - We're not getting better at writing secure code in memory unsafe languages

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Mitigations for Memory Unsafe Languages (1/2)

- Use modern tooling for C and C++ development:
  - Compile C code with, at least, **`clang –W -Wall -Werror`**
    - Review documentation to find additional warnings it makes sense to enable
    - Fix *all* warnings – let the compiler help you debug your code
  - Use **`clang`** static analysis tools during debugging:
    - https://clang.llvm.org/docs/AddressSanitizer.html
    - https://clang.llvm.org/docs/MemorySanitizer.html
    - https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
    - https://clang.llvm.org/docs/ThreadSafetyAnalysis.html
    - https://clang.llvm.org/docs/ThreadSanitizer.html
    - These have very high overhead, but catch many memory and thread safety problems

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Mitigations for Memory Unsafe Languages (2/2)

- Use modern C++ language features:
  - I advise against C++ programming – language is too complex to understand
    - Too many features have been added over time, and too few removed; code mixes old and new features
    - Very few people know the whole language
    - In hindsight, some C++ defaults were inappropriate
    - Modern C++ addresses many of these issues – but must retain backwards compatibility
      - Rust adopts many lessons learned in the development of C++
      - And can be simpler, since doesn't need worry about compatibility with legacy code
  - But if you have to use C++, modernise the code base, to use newer – safer – language features and idioms, where possible

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Gradually Rewrite code into a Memory Safe Language

- Consider rewriting the most critical sections of the code in a memory safe language:
  - Rust can call C functions directly, and can be compiled into a library that can be directly linked with C or C++
    - https://doc.rust-lang.org/nomicon/ffi.html
    - https://unhandledexpression.com/rust/2017/07/10/why-you-should-actually-rewrite-it-in-rust.html
    - Possible – if not necessarily easy – to do a gradual rewrite of C or C++ into a safe language, while keeping the application usable and building at each stage
- Objective-C and Swift can similarly be combined in the Apple ecosystem, to do a gradual rewrite of an application into Swift
- Difficult when languages have very different amount of runtime
  - e.g., rewriting parts of a C program in Java likely not feasible

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Parsing and LangSec

- Postel's law
- Parsing

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Remotely Exploitable Vulnerabilities

- Writing safe code in unsafe languages is difficult
  - Easy to rationalise each individual bug – “How could anyone write that?”

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
... other checks ...
fail:
  ... buffer frees (cleanups) ...
  return err;
```

https://dwheeler.com/essays/apple-goto-fail.html

  - But, people will continue to make mistakes
  - Memory safe languages eliminate common classes of vulnerability; strong type systems help detect mistakes early

- Remotely exploitable vulnerabilities threaten any networked system – are there particular classes of bug that cause such vulnerabilities?
  - Can we address those *classes* of bug?

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# The Robustness Principle (Postel's Law)

```
At every layer of the protocols, there is a general rule whose
application can lead to enormous benefits in robustness and
interoperability:

       "Be liberal in what you accept, and
        conservative in what you send"

Software should be written to deal with every conceivable
error, no matter how unlikely; sooner or later a packet will
come in with that particular combination of errors and
attributes, and unless the software is prepared, chaos can
ensue.  In general, it is best to assume that the network is
filled with malevolent entities that will send in packets
designed to have the worst possible effect.  This assumption
will lead to suitable protective design, although the most
serious problems in the Internet have been caused by
un-envisaged mechanisms triggered by low-probability events;
mere human malice would never have taken so devious a course!
```

RFC1122

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# The Robustness Principle (Postel's Law)

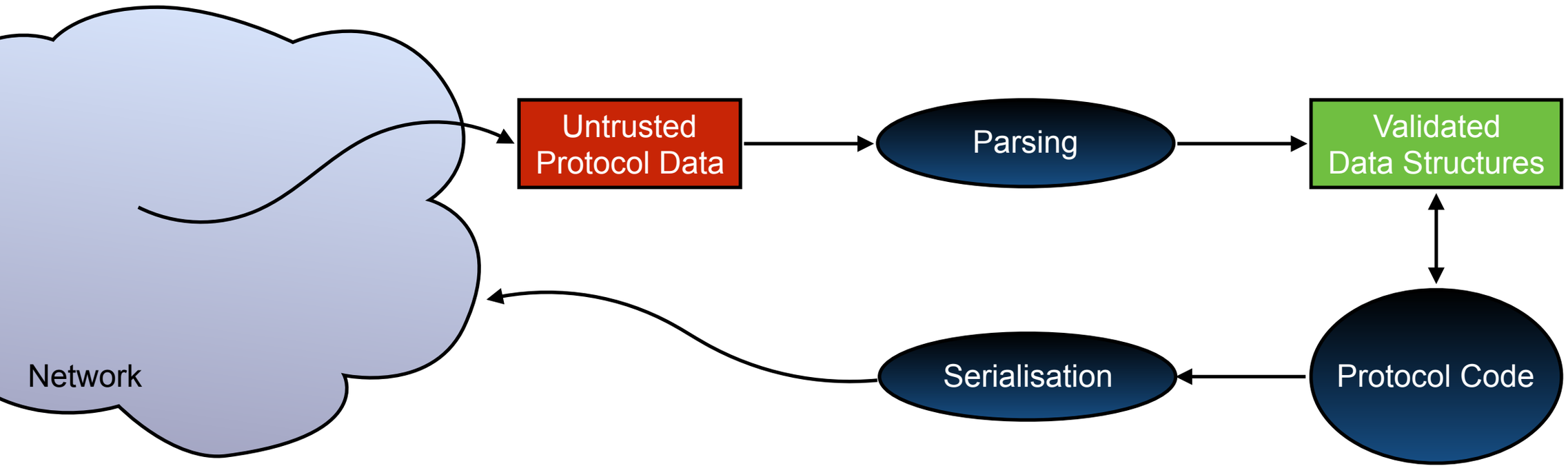
**"Be liberal in what you accept, and conservative in what you send"**

"Postel lived on a network with all his friends.
We live on a network with all our enemies.
Postel was wrong for todays internet."
— *Poul-Henning Kamp*

See also: https://datatracker.ietf.org/doc/draft-iab-protocol-maintenance/

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Parsing Untrusted Input

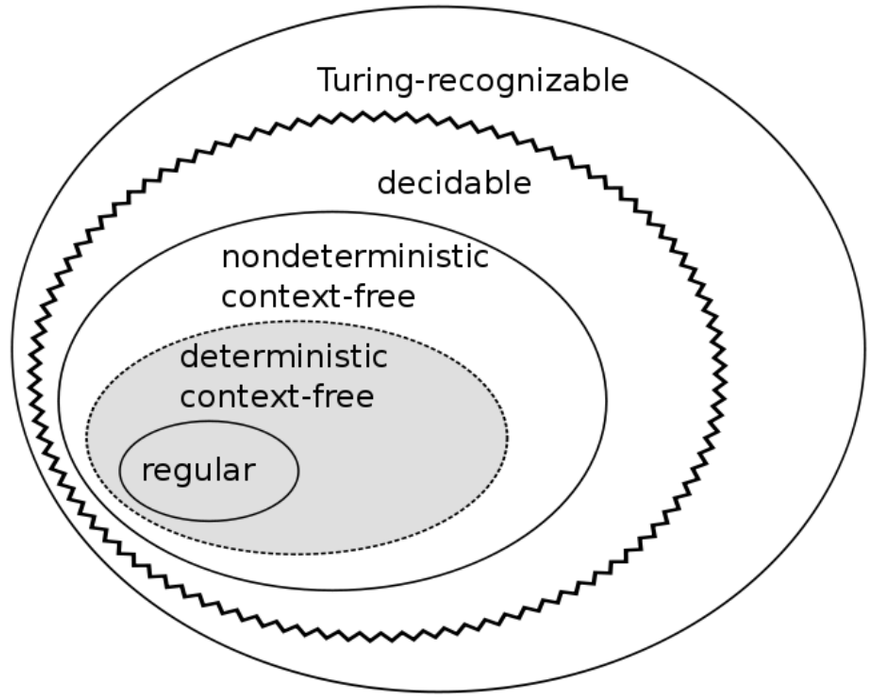



- The input parser is critical to security of a networked system
  - Takes untrusted input from the network
  - Generates validated, strongly typed, data structures that are processed by the rest of the code
- How can we ensure parsers are safe?

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Define Legal Inputs and Failure Responses

- A parser is an automaton, driven by a set of rules and the input data
  - It parses the input according to some grammar
  - If the input does not match the grammar, it fails
- **Formally specify the protocol grammar**
  - Define what is legal, and what is not – write this down in a machine checkable manner
    - Define the grammar in as restrictive a manner as possible – e.g., don't use a Turing-complete parse when a regular expression will suffice
- Specify what happens if the input data does not match the grammar
  - What causes a failure?
  - How is it handled?

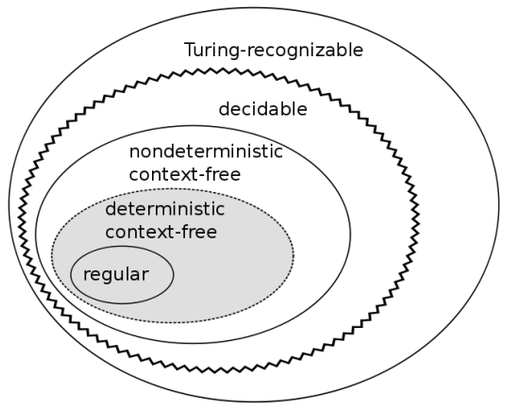



Source: http://langsec.org/papers/Sassaman.pdf

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Generate the Parser (1/3)

- Difficult to reason about ad-hoc, manually written, parsing code
  - It performs low-level bit manipulation, string parsing, etc., all of which are hard to get correct
  - It tends to be poorly structured, hard to follow
- Rather, auto-generate the parsing code:
  - If the input language is regular, use a regular expression
  - If the input language is context free, use a context free grammar
  - If neither of these work, use a more sophisticated parser, with minimal computational power
  - Generate strongly typed data structures, with explicit types to identify different data items
- Focus on parser correctness and readability
  - **Parsing performance matters less than security**




*"Some people, when confronted with a problem, think 'I know, I'll use regular expressions'. Now they have two problems." – Jamie Zawinski*

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Generate the Parser (2/3)

- Use existing, well-tested, parser libraries
  - For Rust code, use **`nom`** or **`combine`**



2017 IEEE Symposium on Security and Privacy Workshops

Writing parsers like it is 2017

Pierre Chifflier
Agence Nationale de la Sécurité
des Systèmes d'Information

Geoffroy Couprie
Clever Cloud

***Abstract*—Despite being known since a long time, memory violations are still a very important cause of security problems in low-level programming languages containing data parsers. We address this problem by proposing a pragmatic solution to fix not only bugs, but classes of bugs. First, using a fast and safe language such as Rust, and then using a parser combinator. We discuss the advantages and difficulties of this solution, and we present two cases of how to implement safe parsers and insert them in large C projects. The implementation is provided as a set of parsers and projects in the Rust language.**

I. Introduction

In 2016, like every year for a long time, memory corruption bugs have been one of the first causes of vulnerabilities of compiled programs [1]. When looking at the C programming language, many errors lead to memory corruption: buffer overflow, use after free, double free, etc. Some of these issues can be complicated to diagnose, and the consequence is that a huge quantity of bugs is hidden in almost all C software.

Any software manipulating untrusted data is particularly exposed: it needs to parse and interpret data that can be controlled by the attacker. Unfortunately, data parsing is often done in a very unsafe way, especially for network protocols and file formats. For example, many bugs were discovered in media parsing libraries in Android [2], leading to the possible remote exploitation of all devices by a simple MMS message. Today, many applications embed a lot of parsers that could be targeted: web browsers like Firefox or Chrome, media players (VLC), document viewers, etc.

Ironically, security tools such as intrusion detection systems or network analyzers suffer from the same problems, making the security tools a possible and interesting target for an attacker. For example, Wireshark had 95 vulnerabilities in 2016 that could crash the application or be exploited.

As a result, most programs written in C are unsafe, especially in the parts of the code parsing data. For example, more than 500 vulnerabilities on XML parsers are listed in the US National Vulnerability Database (NVD). Even for simple formats like JSON, it's hard [3]: some parsers crash, others have bugs, and many of them give different results because of a different interpretation of the specifications.

The Cloudbleed [4] vulnerability, caused by a bug in a HTML parser written in C, caused sensitive data to leak, and is estimated to impact more than 5 million websites. This bug was possible because of the use of an unsafe programming language, despite using a parser generator (Ragel).

In this paper, we propose a pragmatic solution to write robust parsers and significantly improve software security. First, we show how changing the programming language can solve most of the memory-related problems. Second, we show how parser combinators both help prevent bugs and create faster parsers. We then explain how this solution was implemented in two different large C programs, VLC media player and Suricata, to integrate safe parsers by changing only a small amount of code.

II. Current Solutions, and how to go further

*A. Partial and bad solutions*

Many tools, like fuzzing or code audits, come too late in the development process: bugs are already present in the code. Instead, developers should focus on solutions allowing them to prevent bugs during development.

Some are trying to improve quality by integrating automated tests during the development process. The devops trend encourages pushing code into production as fast as possible, minimizing the delay by relying on automated tests to assess security. It is important to be agile and be able to fix bugs very quickly. However, the problem with that approach is that it will catch only known (tested) bugs, and will never be exhaustive: this is more compliance testing than security tests. While it can protect against most regressions, it tends to create a false sense of security.

*B. Reduce Damage*

Hardening methods can be applied to restrict the software to its intended behavior, or limit the consequences in case of successful attacks. While these mechanisms significantly improve the security of the software and the system, they will not fix the initial problem: the restricted application manipulating data is still vulnerable and affected by all the parsing problems. Still, they are interesting, and applications should consider using them as much as possible.

Compiler and OS offer hardening functions that can be used. Some of them can be applied without modifying the software, but the others require a modification or recompilation. We list here only categories of system hardening features.

*a) Compiler functions:* The compiler can apply hardening functions during the compilation: randomization of the address space (ASLR), stack protector, marking sections and the relocation table as read-only, etc. These functions are activated by compilation flags, and will also trigger build errors on some vulnerable code patterns.

© 2017, Pierre Chifflier. Under license to IEEE.
DOI 10.1109/SPW.2017.39

80

IEEE computer society

P. Chifflier and G. Couprie. Writing parsers like it is 2017. In Proceedings of the Workshop on Language-Theoretic Security (LangSec), pages 80–92, San Jose, CA, USA, May 2017. IEEE.

- For C or C++, use **`hammer`**



nom, eating data byte by byte

https://github.com/Geal/nom

Combine: A parser combinator library for Rust

https://github.com/Marwes/combine

Hammer: Parser combinators for binary formats, in C. Yes, in C. What? Don't look at me like that.

https://github.com/UpstandingHackers/hammer

CC BY ND

# Generate the Parser (3/3)

```
#[derive(Debug,PartialEq,Eq)]
pub struct Header {
  pub version: u8,
  pub audio:   bool,
  pub video:   bool,
  pub offset:  u32,
}
```

```
do_parse!(
            tag!("FLV") >>
  version: be_u8       >>
  flags:   be_u8       >>
  offset:  be_u32      >>
  (Header {
      version: version,
      audio:   flags & 4 == 4,
      video:   flags & 1 == 1,
      offset:  offset
  })
)
);
```

- Specify the types into which parsed data is stored
- Describe the parser using an appropriate formal language – example uses **`nom`**
- Generate the parser from that language

Example from: P. Chifflier and G. Couprie. Writing parsers like it is 2017. In Proceedings of the Workshop on Language-Theoretic Security (LangSec), pages 80–92, San Jose, CA, USA, May 2017. IEEE. https://dx.doi.org/10.1109/SPW.2017.39

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Define Parsable Protocols

- If designing network protocols, consider ease of parsing
  - Minimise the amount of state and look-ahead required to parse the data
  - Prefer a predictable, regular, grammar to one that saves bits at the expense of complex parsing
    - Networks get faster, security vulnerabilities remain
    - The benefit of saving a few bits gets less over time
    - The benefit of being easy to parse securely remains

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Parsing and LangSec

- Review “The Bugs We Have to Kill”
  - Is this approach realistic?
  - Can we combine better parsing with modern, strongly typed, languages to improve network security?
  - Is performance good enough?
  - Can we improve the way we design protocols?



SECURITY

## The Bugs We Have to Kill

SERGEY BRATUS, MEREDITH L. PATTERSON, AND ANNA SHUBINA

Sergey Bratus is a Research Associate Professor of computer science at Dartmouth College. He sees state-of-the-art hacking as a distinct research and engineering discipline that, although not yet recognized as such, harbors deep insights into the nature of computing. He has a PhD in mathematics from Northeastern University and worked at BBN Technologies on natural language processing research before coming to Dartmouth.
sergey@cs.dartmouth.edu

Meredith L. Patterson is the founder of Upstanding Hackers. She developed the first language-theoretic defense against SQL injection in 2005 as a PhD student at the University of Iowa and has continued expanding the technique ever since. She lives in Brussels, Belgium.
mlp@upstandinghackers.com

Anna Shubina is a Research Associate at the Dartmouth Institute for Security, Technology, and Society and maintains the CRAWDAD.org repository of traces and data for all kinds of wireless and sensor network research. She was the operator of Dartmouth's Tor node when the Tor network had about 30 nodes total.
ashubina@cs.dartmouth.edu

The code that parses inputs is the first and often the only protection for the rest of a program from malicious inputs. No programmer can afford to verify every implied condition on every line of code—even if this were possible to implement without slowing execution to a crawl. The parser is the part that is supposed to create a world for the rest of the program where all these implied conditions are true and need not be explicitly checked at every turn. Sadly, this is exactly where most parsers fail, and the rest of the program fails with them. In this article, we explain why parsers continue to be such a problem, as well as point to potential solutions that can kill large classes of bugs.

To do so, we are going to look at the problem from the computer science theory angle. Parsers, being input-consuming machines, are quite close to the theory's classic computing models, each one an input-consuming machine: finite automata, pushdown automata, and Turing machines. The latter is our principal model of general-purpose programming, the computing model with the ultimate power and flexibility. Yet this high-end power and flexibility come with a high price, which Alan Turing demonstrated (and to whose proof we owe our very model of general-purpose programming): our inability to predict, by any general static analysis algorithm, how programs for it will execute.

Yet most of our parsers are just a layer on top of this totally flexible computing model. It is not surprising, then, that without carefully limiting our parsers' design and code to much simpler models, we are left unable to prove these input-consuming machines secure. This is a powerful argument for making parsers and their input formats and protocols simpler, so that securing them does not require having to solve undecidable problems!

### Parsers, Parsers Everywhere

To quote Koprowski and Binsztok [1]:

> Parsing is of major interest in computer science. Classically discovered by students as the first step in compilation, parsing is present in almost every program which performs data-manipulation. For instance, the Web is built on parsers. The HyperText Transfer Protocol (HTTP) is a parsed dialog between the client, or browser, and the server. This protocol transfers pages in HyperText Markup Language (HTML), which is also parsed by the browser. When running web-applications, browsers interpret JavaScript programs which, again, begins with parsing. Data exchange between browser(s) and server(s) uses languages or formats like XML and JSON. Even inside the server, several components (for instance the trio made of the HTTP server Apache, the PHP interpreter and the MySQL database) often manipulate programs and data dynamically; all require parsers.

So do the lower layers of the network stack down to the IP and the link layer protocols, and also other OS parts such as the USB drivers [2] (and even the hardware: turning PHY layer symbol streams into frames is parsing, too!). For all of these core protocols, we add, their parsers have had a long history of failures, resulting in an Internet where any site, program, or system that receives untrusted input can be presumed compromised.



S. Bratus, M. L. Patterson, and A. Shubina. The bugs we have to kill. ;login:, 40(4):4–10, August 2015. http://langsec.org/papers/the-bugs-we-have-to-kill.pdf

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Modern Type Systems and Security

- Make assumptions explicit
- Eliminate undefined behaviour

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Causes of Security Vulnerabilities

- Security vulnerabilities generally caused by persuading a program to do something that the programmer did not expect
  - Write past the end of a buffer
  - Treat user input as executable
  - Confuse a permission check
  - …
- Violate an assumption in the code

Colin Perkins | https://csperkins.org/ | Copyright © 2019

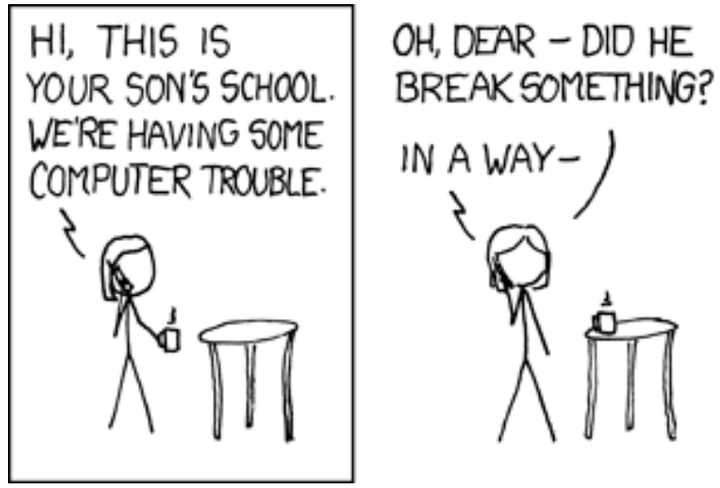
# Causes of Security Vulnerabilities

- Strong typing makes assumptions explicit
  - Use explicit types rather than generic types
  - Define safe conversion functions
  - Use phantom types where necessary, to apply semantic tags to data

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Prefer Explicit Types

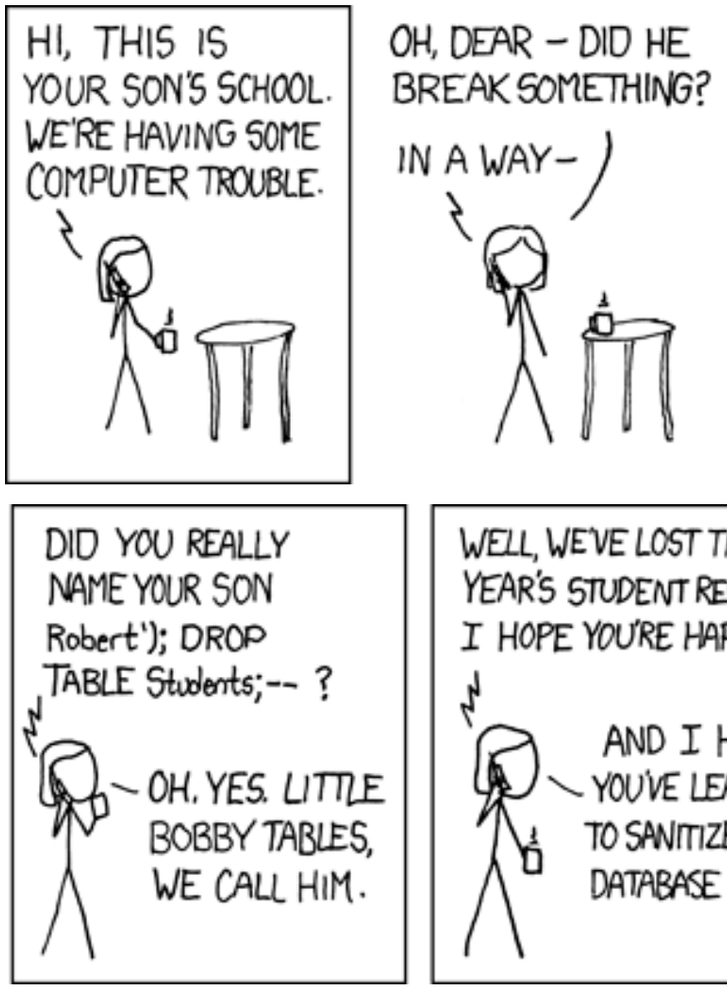
- Vulnerabilities come from inconsistent data processing:
  - e.g., passing un-sanitised user entered data to a function that expects valid SQL
    - A **`StudentName`** and an **`SqlString`** are different; give them different types so compiler can catch inconsistent usage
      - Certain characters must be escaped before an arbitrary student name can be safely stored in an SQL database
- If **`String`** used everywhere, the programmer must manually check for consistency
  - Easy to make mistakes
  - If all the types are the same, the compiler can't help




Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Convert Data Carefully

- Explicit types require type conversions
  - Enforce security boundaries
  - Untrusted user input → escaped, safe, internal form; validate input before using it
  - Ensure only legal conversions occur




Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Use Phantom Types to Add Semantic Tags

- A phantom type parameter is one that doesn't show up at runtime, but is checked at compile time
  - In Rust, a **`struct`** with no fields has a type but is zero sized
  - Useful as type parameters to add semantic tags to data:

```
struct UserInput;  // As received from the user
struct Sanitised;  // After HTML codes have been escaped

fn sanitise_html(html : Html<UserInput>) -> Html<Sanitised> {
   …
}
```

  - Useful to represent states in a state machine

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# No Silver Bullet

Memory safety and strong typing won’t eliminate security vulnerabilities



But, used carefully, they eliminate certain classes of vulnerability, and make others less likely by making hidden assumptions – and their consequences – visible

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Liability and Ethics

## ACM code of ethics and professional conduct:

### 1.2 Avoid harm.

In this document, "harm" means negative consequences, especially when those consequences are significant and unjust. Examples of harm include unjustified physical or mental injury, unjustified destruction or disclosure of information, and unjustified damage to property, reputation, and the environment. This list is not exhaustive.

Well-intended actions, including those that accomplish assigned duties, may lead to harm. When that harm is unintended, those responsible are obliged to undo or mitigate the harm as much as possible. Avoiding harm begins with careful consideration of potential impacts on all those affected by decisions. When harm is an intentional part of the system, those responsible are obligated to ensure that the harm is ethically justified. In either case, ensure that all harm is minimized.

To minimize the possibility of indirectly or unintentionally harming others, computing professionals should follow generally accepted best practices unless there is a compelling ethical reason to do otherwise. Additionally, the consequences of data aggregation and

https://www.acm.org/binaries/content/assets/about/acm-code-of-ethics-and-professional-conduct.pdf

Security vulnerabilities and software failures routinely cause harm – can you justify your professional practice?

Colin Perkins | https://csperkins.org/ | Copyright © 2019

# Summary

- Memory safety
- Parsing and LangSec
- Modern Type Systems and Security
- Ethics and Liability

Colin Perkins | https://csperkins.org/ | Copyright © 2019