Error Handling

CS110L
Jan 19, 2022

Logistics

- Congrats on making it almost 1/3 through the quarter! 🎉
- Week 2 exercises due yesterday — hopefully you got some practice doing stuff in Rust!
  - Let me know if you need more time.
  - Fill out check-in survey linked on week 2 exercise handout!!
- Week 3 exercises: out today, due next **Thursday**.
  - No week 4 exercises.
- Lecture notes:
  - Lecture notes from last week: practice with ownership, including deeper explanation of “why Rust avoids iterator invalidation” example.
  - Lecture notes from today: more practice with error handling (today’s topic).

Logistics

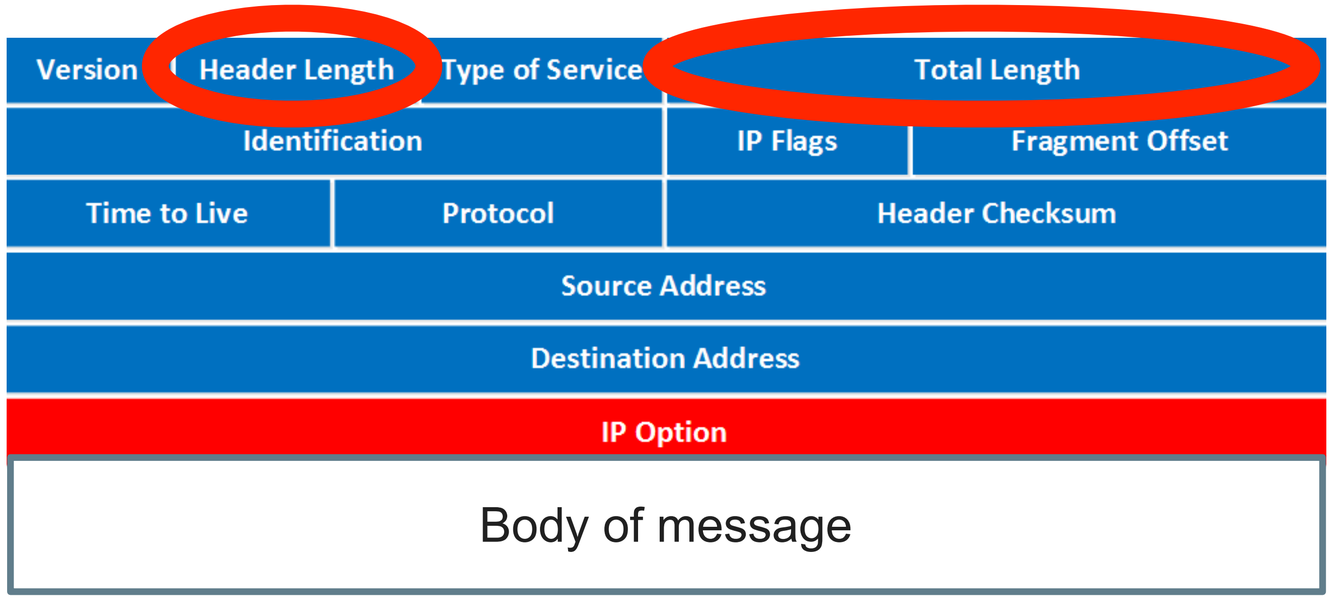
- Transition back to in-person will be on the same schedule/policies as CS110 and undergrad non-lab classes.
- Will have lecture videos from last year posted on Canvas weekly.
  - Do not share them beyond this class!!
- Communicate with me if you want/need a fully remote way to engage with the class, if you're going to be absent for more than a day or two, etc.
- Will have both remote and in-person office hours (& always happy to meet outside of OH if needed!). Will be posted on 110L calendar.

Error handling

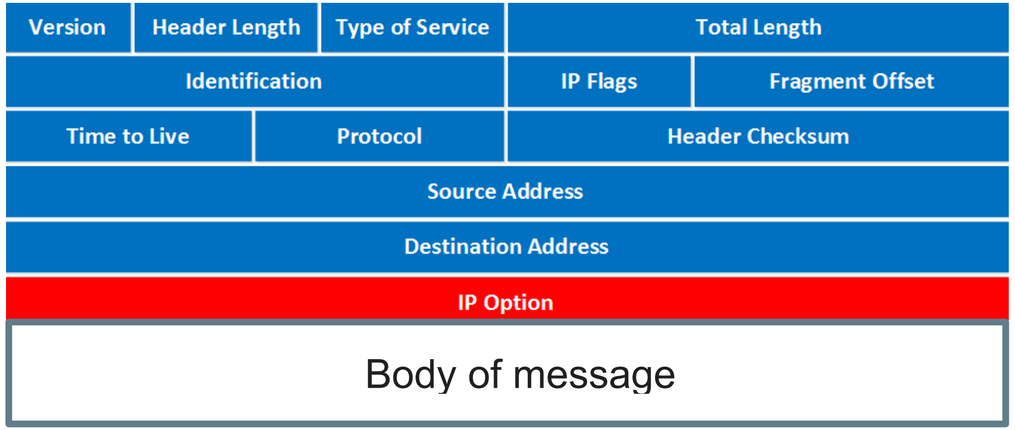
More Remote Code Execution

- Imagine a server receives messages from the network
  - Like all messages that travel over the Internet, it's encapsulated in an IP (IPv4) header
  - IP header *can* be variable length. Length of IP header [supposed to be] specified in "header length" field.
  - Length of whole message [supposed to be] specified in "total length" field.

- *Note that anyone (e.g., an attacker!) can populate these fields*



More Remote Code Execution




```
struct message {
   ipv4_hdr iphdr;
   ipv4_options[MAX_IP_OPTIONS] opts;
   char[MAX_DATA_LEN] data;
}
```

```
/* Given: read-only copy of entire message, read in from
   the network. */
void* process_and_return_data(const struct message *msg) {

    // Allocate space for local, mutable copy.
    void *local_copy = malloc(get_len(msg));

    // Copy only the body of the message
    memcpy(local_copy + get_hdr_len(msg->iphdr),
           msg + get_hdr_len(msg->iphdr),
           length_of_body);

    process_data(local_copy + get_hdr_len(msg->iphdr));

    // Copy in IP hdr
    memcpy(local_copy, msg, get_hdr_len(msg->iphdr));

    return local_copy;
}
```

More Remote Code Execution

The **malloc**() function allocates *size* bytes and returns a pointer
to the allocated memory. *The memory is not initialized*. If *size*
is 0, then **malloc**() returns either NULL, or a unique pointer
value that can later be successfully passed to **free**().

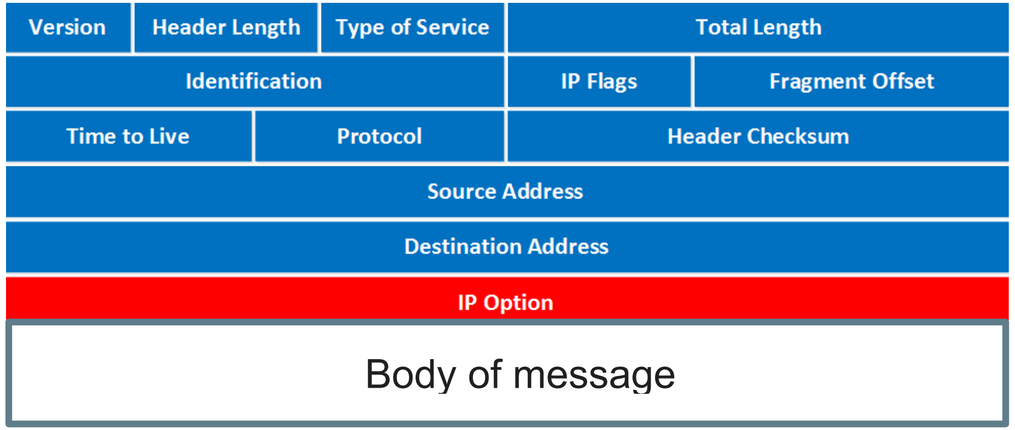
type. On error, these functions return NULL. NULL may also be
returned by a successful call to **malloc**() with a *size* of zero, or

calloc(), **malloc**(), **realloc**(), and **reallocarray**() can fail with
the following error:

ENOMEM Out of memory. Possibly, the application hit the
RLIMIT_AS or **RLIMIT_DATA** limit described in getrlimit(2).

https://man7.org/linux/man-pages/man3/free.3.html

More Remote Code Execution




```
struct message {
    ipv4_hdr iphdr;
    ipv4_options[MAX_IP_OPTIONS] opts;
    char[MAX_DATA_LEN] data;
}
```

```
/* Given: read-only copy of entire message, read in from
   the network. */
void* process_and_return_data(const struct message *msg) {

    // Allocate space for local, mutable copy.
    void *local_copy = malloc(get_len(msg));

    // Copy only the body of the message
    memcpy(local_copy + get_hdr_len(msg->iphdr),
           msg + get_hdr_len(msg->iphdr),
           length_of_body);

    process_data(local_copy + get_hdr_len(msg->iphdr));

    // Copy in IP hdr
    memcpy(local_copy, msg, get_hdr_len(msg->iphdr));

    return local_copy;
}
```

Key insight: `malloc` could fail and return NULL

`local_copy + [value]` could be… anything.

Similar things have happened…

https://cve.mitre.org/cve/search_cve_list.html

CVE-2009-3448 Learn more at National Vulnerability Database (NVD)
• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

npvmgr.exe in BakBone NetVault Backup 8.22 Build 29 allows remote attackers to cause a denial of service (daemon crash) via a packet to (1) TCP or (2) UDP port 20031 with a large value in an unspecified size field, which is not properly handled in a malloc operation. NOTE: some of these details are obtained from third party information.

CVE-2021-34405	NVIDIA Linux distributions contain a vulnerability in TrustZone’s TEE_Malloc function, where an unchecked return value causing a null pointer dereference may lead to denial of service.
CVE-2021-29605	TensorFlow is an end-to-end open source platform for machine learning. The TFLite code for allocating `TFLiteIntArray`s is vulnerable to an integer overflow issue(https://github.com/tensorflow/tensorflow/blob/4ceffae632721e52bf3501b736e4fe9d1221cdfa/tensorflow/lite/c/common.c#L24-L27). An attacker can craft a model such that the `size` multiplier is so large that the return value overflows the `int` datatype and becomes negative. In turn, this results in invalid value being given to `malloc`(https://github.com/tensorflow/tensorflow/blob/4ceffae632721e52bf3501b736e4fe9d1221cdfa/tensorflow/lite/c/common.c#L47-L52). In this case, `ret->size` would dereference an invalid pointer. The fix will be included in TensorFlow 2.5.0. We will also cherrypick this commit on TensorFlow
CVE-2020-7105	async.c and dict.c in libhiredis.a in hiredis through 0.14.0 allow a NULL pointer dereference because malloc return values are unchecked.
CVE-2021-31890	length of an TCP payload (set in the IP header) is unchecked. This may lead to various side effects, including Information Leak and Denial-of-Service conditions, depending on the network buffer organization in memory. (FSMD-2021-0017)
CVE-2017-8395	The Binary File Descriptor (BFD) library (aka libbfd), as distributed in GNU Binutils 2.28, is vulnerable to an invalid write of size 8 because of missing a malloc() return-value check to see if memory had actually been allocated in the _bfd_generic_get_section_contents function. This vulnerability causes programs that conduct an analysis of binary programs using the libbfd library, such as objcopy, to crash.
CVE-2014-8241	XRegion in TigerVNC allows remote VNC servers to cause a denial of service (NULL pointer dereference) by leveraging failure to check a malloc return value, a similar issue to CVE-2014-6052.

Issues

- Lack of proper error handling
- Use of NULL in place of a real value

Important note but not really related to what we're talking about today: you should never ever EVER trust values that come from the network!

```
/* Given: read-only copy of entire message, read in from
   the network. */
void* process_and_return_data(const struct message *msg) {

    // Allocate space for local, mutable copy.
    void *local_copy = malloc(get_len(msg));

    // Copy only the body of the message
    memcpy(local_copy + get_hdr_len(msg->iphdr),
           msg + get_hdr_len(msg->iphdr),
           length_of_body);

    process_data(local_copy + get_hdr_len(msg->iphdr));

    // Copy in IP hdr
    memcpy(local_copy, msg, get_hdr_len(msg->iphdr));

    return local_copy;
}
```

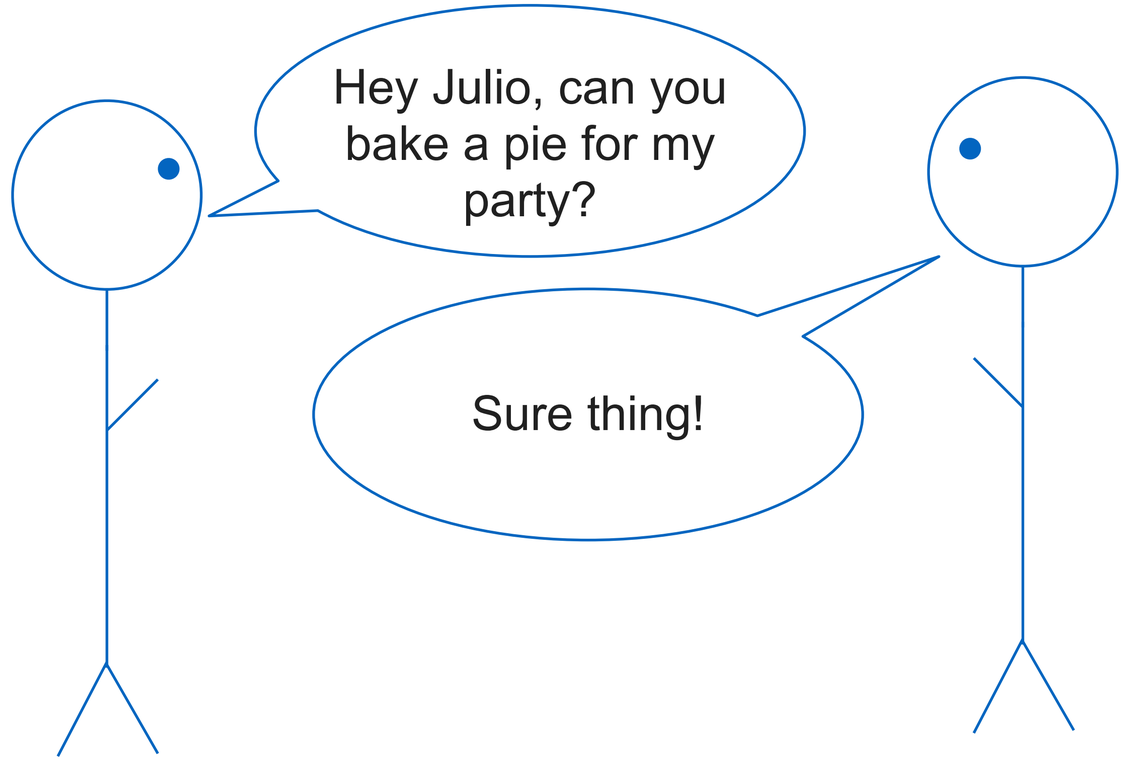
Handling errors

Error handling in C

- If a function might encounter an error, its return type is made to be `int` (or sometimes `void*`).
- If the function is successful, it returns `0`. Otherwise, if an error is encountered, it returns `-1`. (If the function is returning a pointer, it returns a valid pointer in the success case, or `NULL` if an error occurs.)
- The function that encountered the error sets the global variable `errno` to be an integer indicating what went wrong. If the caller sees that the function returned `-1` or `NULL`, it can check `errno` to see what error was encountered

```
#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Arg list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD          10      /* No child processes */
#define EAGAIN          11      /* Try again */
#define ENOMEM          12      /* Out of memory */
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define ENOTBLK         15      /* Block device required */
#define EBUSY           16      /* Device or resource busy */
#define EEXIST          17      /* File exists */
#define EXDEV           18      /* Cross-device link */
#define ENODEV          19      /* No such device */
#define ENOTDIR         20      /* Not a directory */
#define EISDIR          21      /* Is a directory */
#define EINVAL          22      /* Invalid argument */
#define ENFILE          23      /* File table overflow */
#define EMFILE          24      /* Too many open files */
#define ENOTTY          25      /* Not a typewriter */
#define ETXTBSY         26      /* Text file busy */
#define EFBIG           27      /* File too large */
#define ENOSPC          28      /* No space left on device */
#define ESPIPE          29      /* Illegal seek */
#define EROFS           30      /* Read-only file system */
#define EMLINK          31      /* Too many links */
#define EPIPE           32      /* Broken pipe */
#define EDOM            33      /* Math argument out of domain of func */
#define ERANGE          34      /* Math result not representable */
#define EDEADLK         35      /* Resource deadlock would occur */
#define ENAMETOOLONG    36      /* File name too long */
#define ENOLCK          37      /* No record locks available */
#define ENOSYS          38      /* Function not implemented */
#define ENOTEMPTY       39      /* Directory not empty */
#define ELOOP           40      /* Too many symbolic links encountered */
#define EWOULDBLOCK     EAGAIN  /* Operation would block */
#define ENOMSG          42      /* No message of desired type */
#define EIDRM           43      /* Identifier removed */
#define ECHRNG          44      /* Channel number out of range */
#define EL2NSYNC        45      /* Level 2 not synchronized */
#define EL3HLT          46      /* Level 3 halted */
#define EL3RST          47      /* Level 3 reset */
#define ELNRNG          48      /* Link number out of range */
#define EUNATCH         49      /* Protocol driver not attached */
#define EL2HLT          51      /* Level 2 halted */
#define EBADE           52      /* Invalid exchange */
#define EBADR           53      /* Invalid request descriptor */
#define EXFULL          54      /* Exchange full */
#define ENOANO          55      /* No anode */
#define EBADRQC         56      /* Invalid request code */
#define EBADSLT         57      /* Invalid slot */
#define EBFONT          59      /* Bad font file format */
#define ENOSTR          60      /* Device not a stream */
#define ENODATA         61      /* No data available */
#define ETIME           62      /* Timer expired */
#define ENOSR           63      /* Out of streams resources */
#define ENONET          64      /* Machine is not on the network */
#define ENOPKG          65      /* Package not installed */
#define EREMOTE         66      /* Object is remote */
#define ENOLINK         67      /* Link has been severed */
#define EADV            68      /* Advertise error */
#define ESRMNT          69      /* Srmount error */
#define ECOMM           70      /* Communication error on send */
#define EPROTO          71      /* Protocol error */
#define EMULTIHOP       72      /* Multihop attempted */
#define EDOTDOT         73      /* RFS specific error */
#define EBADMSG         74      /* Not a data message */
#define EOVERFLOW       75      /* Value too large for defined data type */
#define ENOTUNIQ        76      /* Name not unique on network */
#define EBADFD          77      /* File descriptor in bad state */
#define EREMCHG         78      /* Remote address changed */
#define ELIBACC         79      /* Can not access a needed shared library */
#define ELIBBAD         80      /* Accessing a corrupted shared library */
#define ELIBSCN         81      /* .lib section in a.out corrupted */
#define ELIBMAX         82      /* Attempting to link in too many shared libraries */
#define ELIBEXEC        83      /* Cannot exec a shared library directly */
#define EILSEQ          84      /* Illegal byte sequence */
#define ERESTART        85      /* Interrupted system call should be restarted */
#define ESTRPIPE        86      /* Streams pipe error */
#define EUSERS          87      /* Too many users */
#define ENOTSOCK        88      /* Socket operation on non-socket */
#define EDESTADDRREQ    89      /* Destination address required */
#define EMSGSIZE        90      /* Message too long */
#define EPROTOTYPE      91      /* Protocol wrong type for socket */
#define ENOPROTOOPT     92      /* Protocol not available */
#define EPROTONOSUPPORT 93      /* Protocol not supported */
#define ESOCKTNOSUPPORT 94      /* Socket type not supported */
#define EOPNOTSUPP      95      /* Operation not supported on transport endpoint */
#define EPFNOSUPPORT    96      /* Protocol family not supported */
#define EAFNOSUPPORT    97      /* Address family not supported by protocol */
#define EADDRINUSE      98      /* Address already in use */
#define EADDRNOTAVAIL   99      /* Cannot assign requested address */
...
```

Error handling in C



int main()

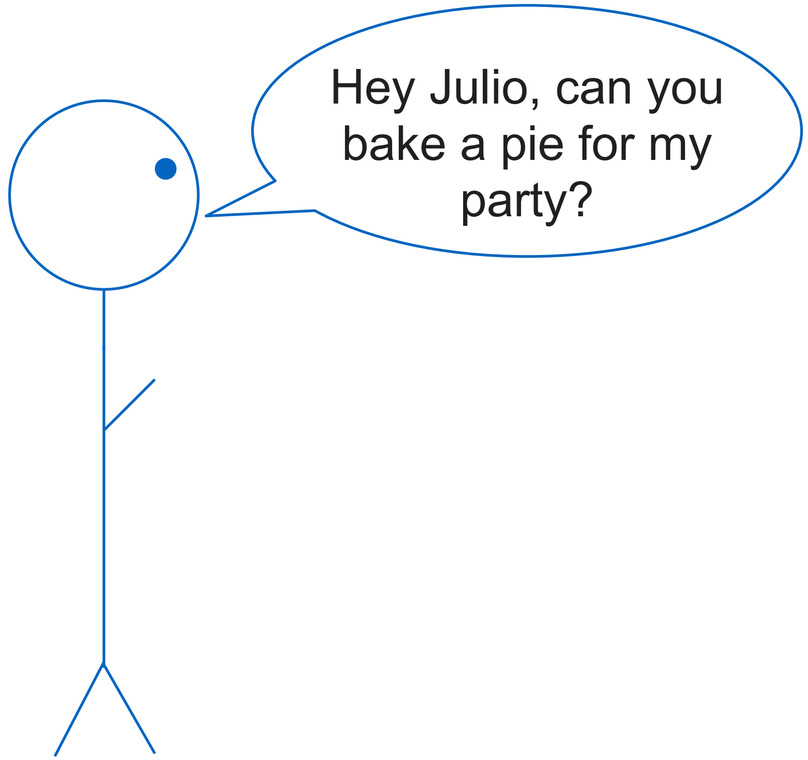
julio:

```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```

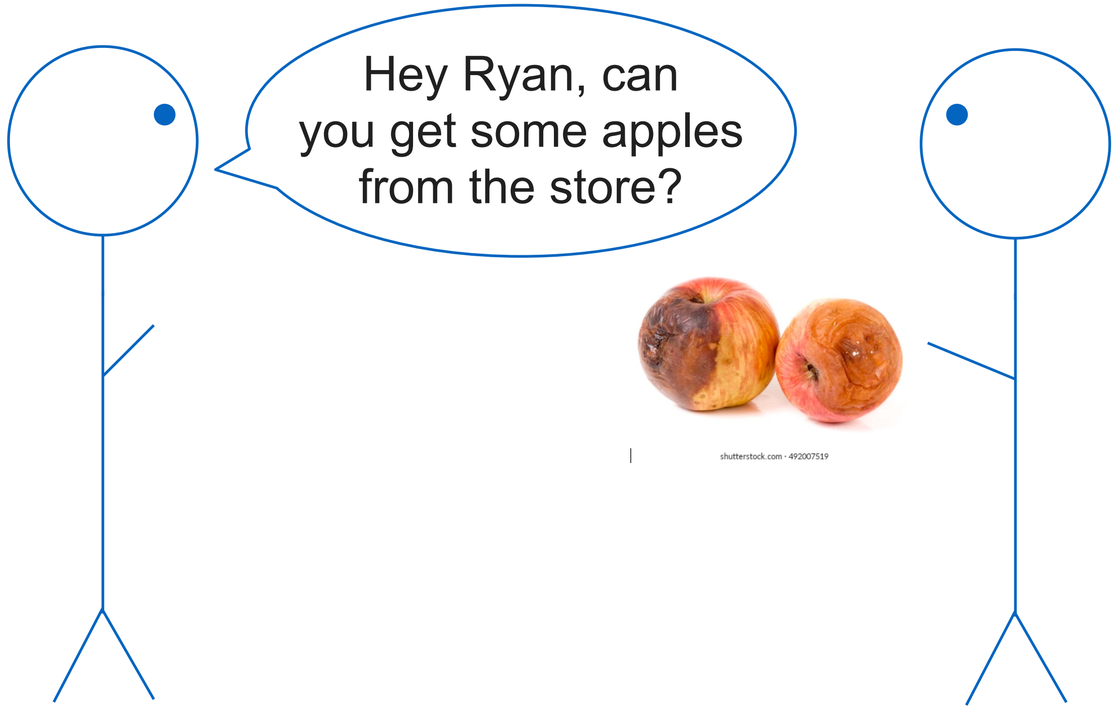
ryan:

```
struct apple *get_apples()
```

Error handling in C



int main()



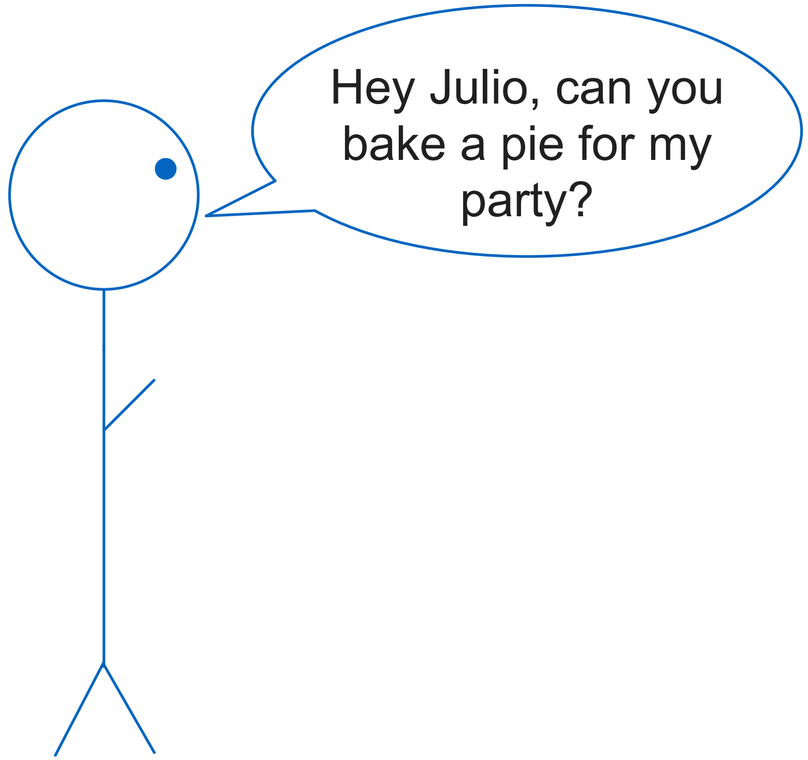
julio:

```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

Error handling in C



int main()

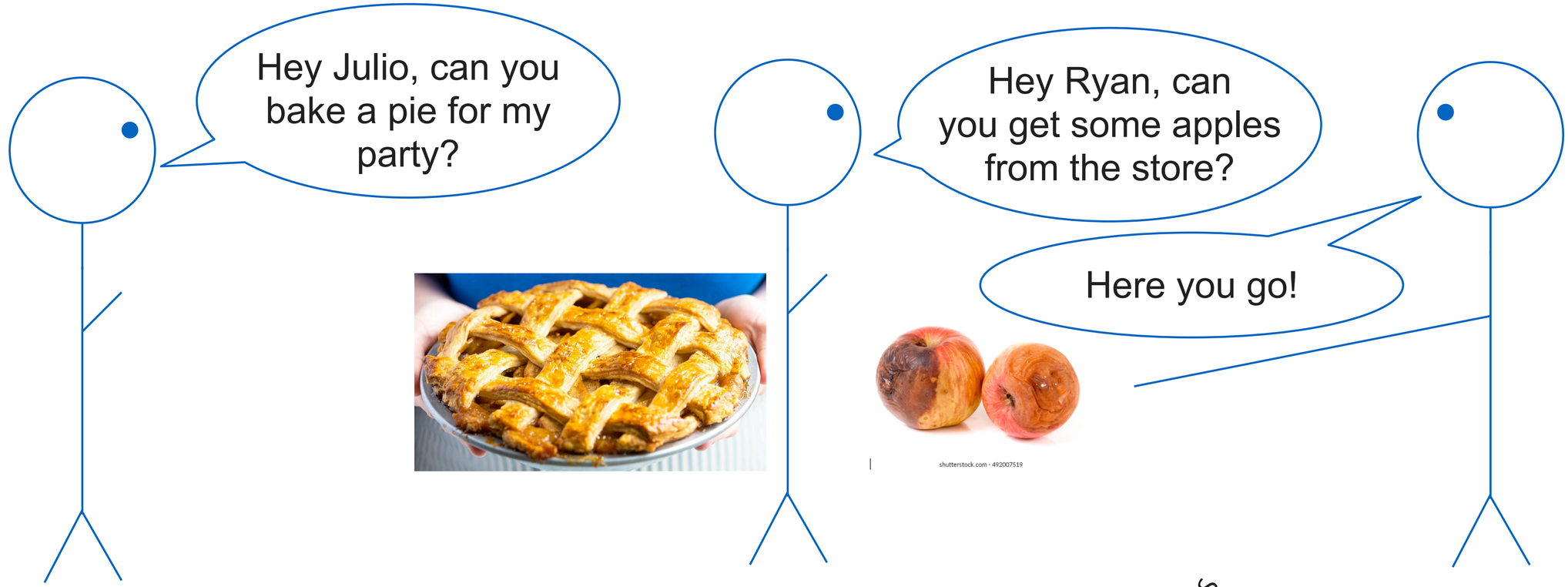
julio:

```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```


ryan:

```
struct apple *get_apples()
```

Error handling in C



int main()

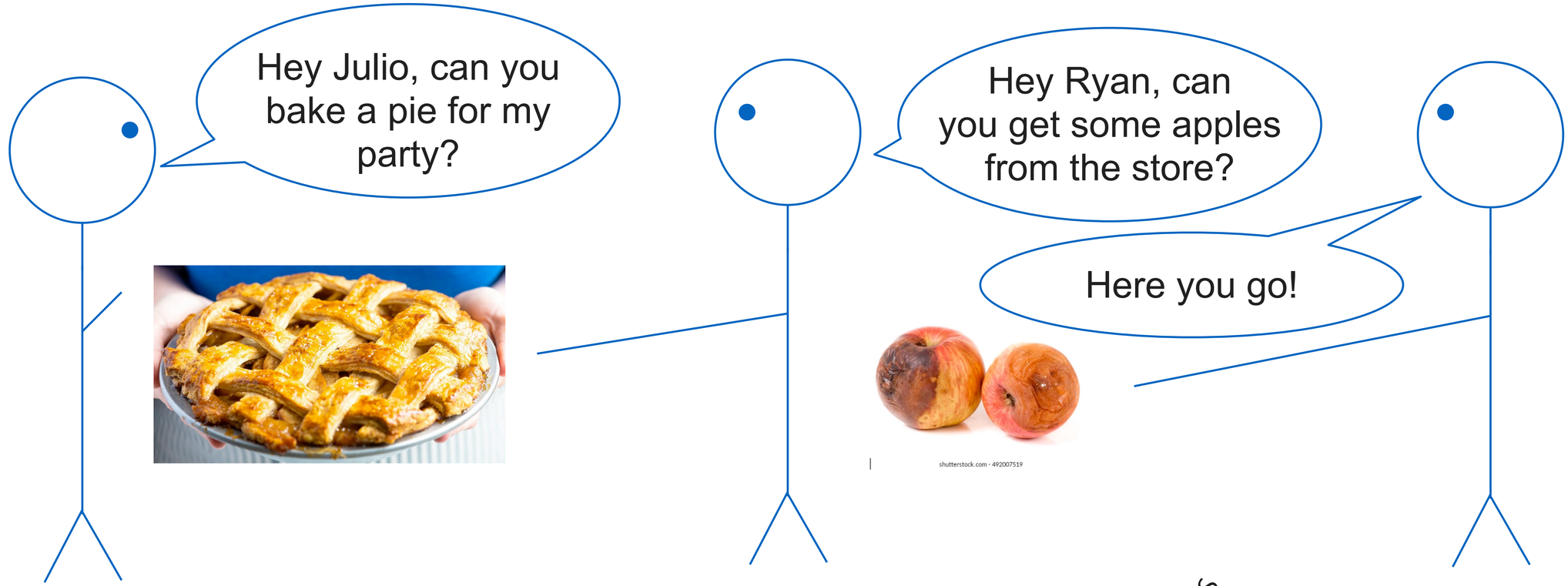
julio:

```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

Error handling in C



int main()

julio:

```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

Broken code from earlier

```
/* Given: read-only copy of entire message, read in from
   the network. */
void* process_and_return_data(const struct message *msg) {

    // Allocate space for local, mutable copy.
    void *local_copy = malloc(get_len(msg));

    // Copy only the body of the message
    memcpy(local_copy + get_hdr_len(msg->iphdr),
           msg + get_hdr_len(msg->iphdr),
           length_of_body);

    process_data(local_copy + get_hdr_len(msg->iphdr));

    // Copy in IP hdr
    memcpy(local_copy, msg, get_hdr_len(msg->iphdr));

    return local_copy;
}
```

Missing error check! 💣

CVE-2015-8812

- Critical Linux kernel vulnerability: by sending a malformed network packet, a remote attacker could execute arbitrary code in the kernel
- A set of kernel networking functions were returning -1 for error, 0 for success, but also other values for “warnings”
  - Returned NET_XMIT_CN (defined to be 2) when congestion was detected
- Code calling these functions saw nonzero return code and assumed there was a network error
- Freed memory that was still being used for the network. Use-after-free + double free!

The fix

```
--- a/drivers/infiniband/hw/cxgb3/iwch_cm.c
+++ b/drivers/infiniband/hw/cxgb3/iwch_cm.c
@@ -149,7 +149,7 @@ static int iwch_l2t_send(struct t3cdev *tdev, struct sk_buff *skb, struct
l2t_en
      error = l2t_send(tdev, skb, l2e);
      if (error < 0)
           kfree_skb(skb);
-     return error;
+     return error < 0 ? error : 0;
 }
```

Key insight

- Different return value possibilities to indicate success + different kinds of errors (this is really common)
- Documented in (e.g.) documentation pages and/or header comments
- All of these are just integers
- Caller must remember to handle all cases

Proper C error checking is ugly

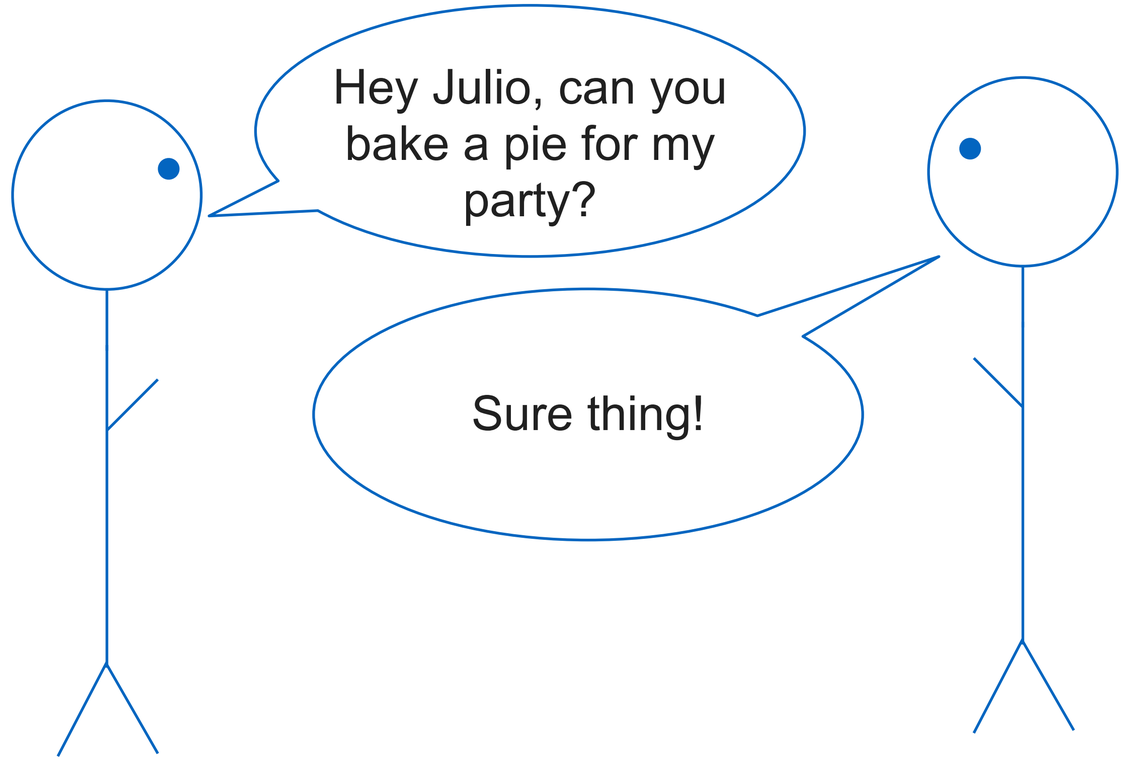
- Programmer must remember whether the function they are calling might return an error
  - Places immense burden on the programmer to remember how each specific function works
  - Might be multiple kinds of errors — must handle all of them!
- After every function call that might return an error, must check whether an error occurred and handle it **correctly**
  - This isn't good enough (why not?):

```
void *buf = malloc();
if (buf == NULL) {
    perror("error allocating memory");
}
memcpy(buf + offset, src, size);
```

- Handling specific errors using `errno` can produce an error-prone mess of `if` statements
  - Sometimes function documentation does not even properly document what errors might be returned

Error-handling in C++
(and many other languages)

Error handling in C++: Exceptions



int main()

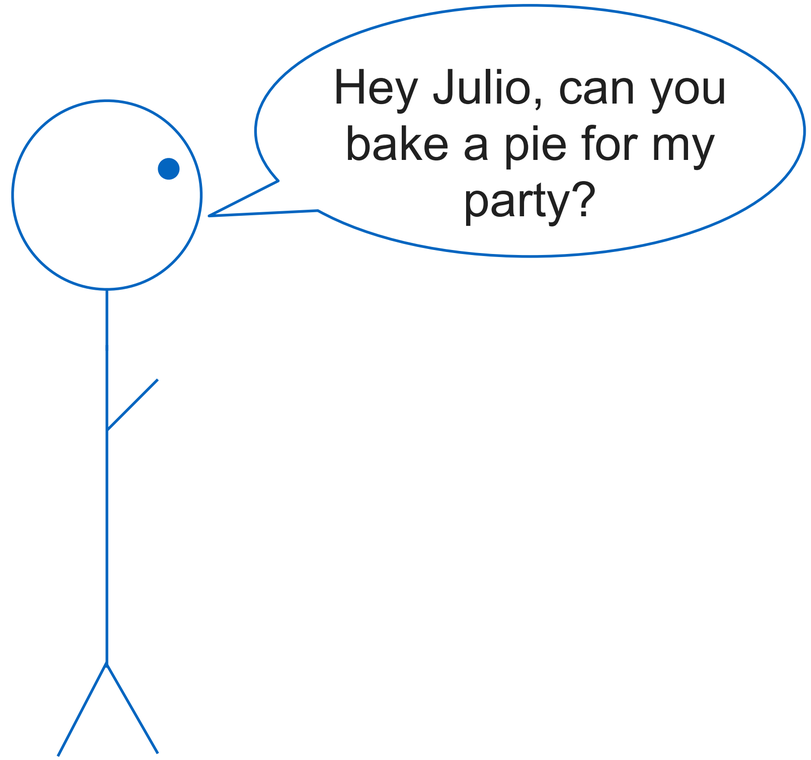
julio:

```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```

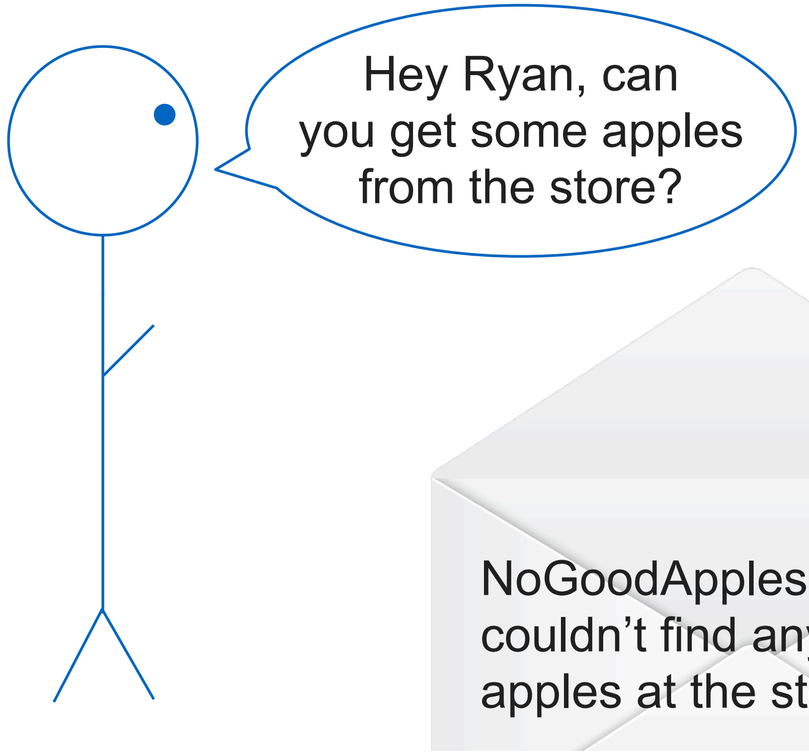
ryan:

```
struct apple *get_apples()
```

Error handling in C++: Exceptions



int main()



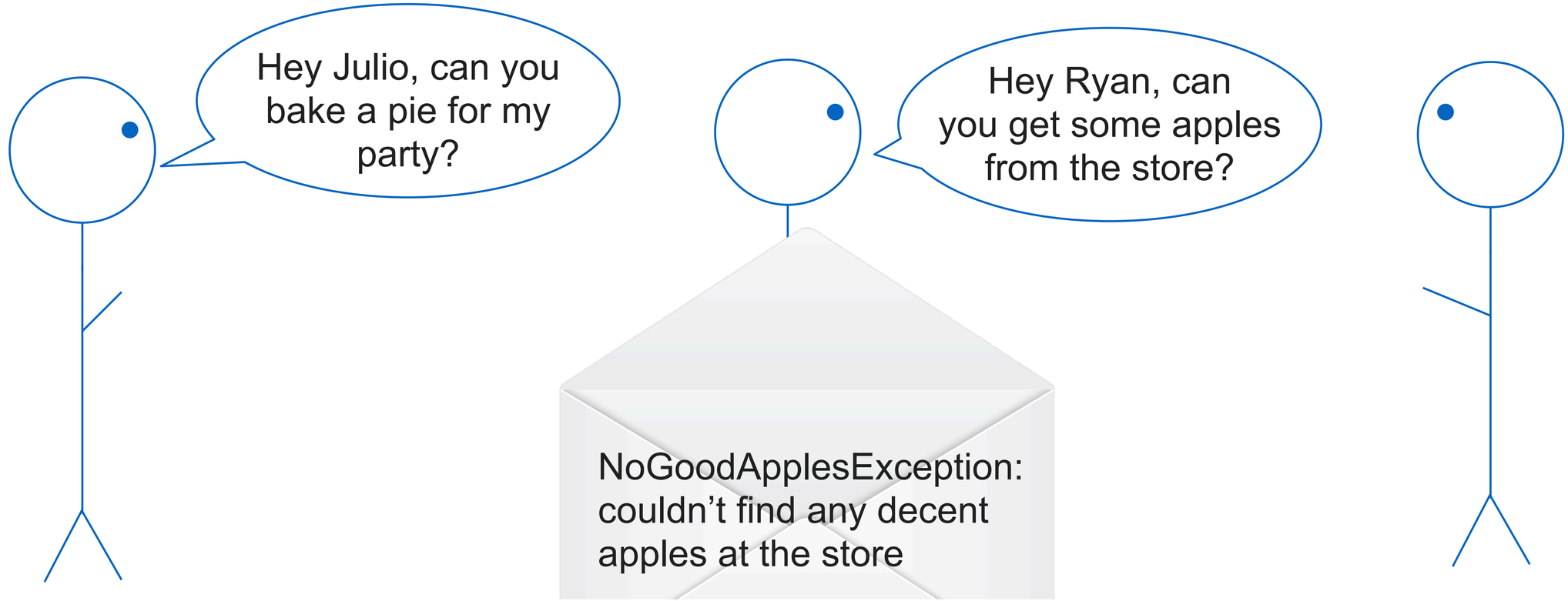
julio:

```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

Error handling in C++: Exceptions



int main()

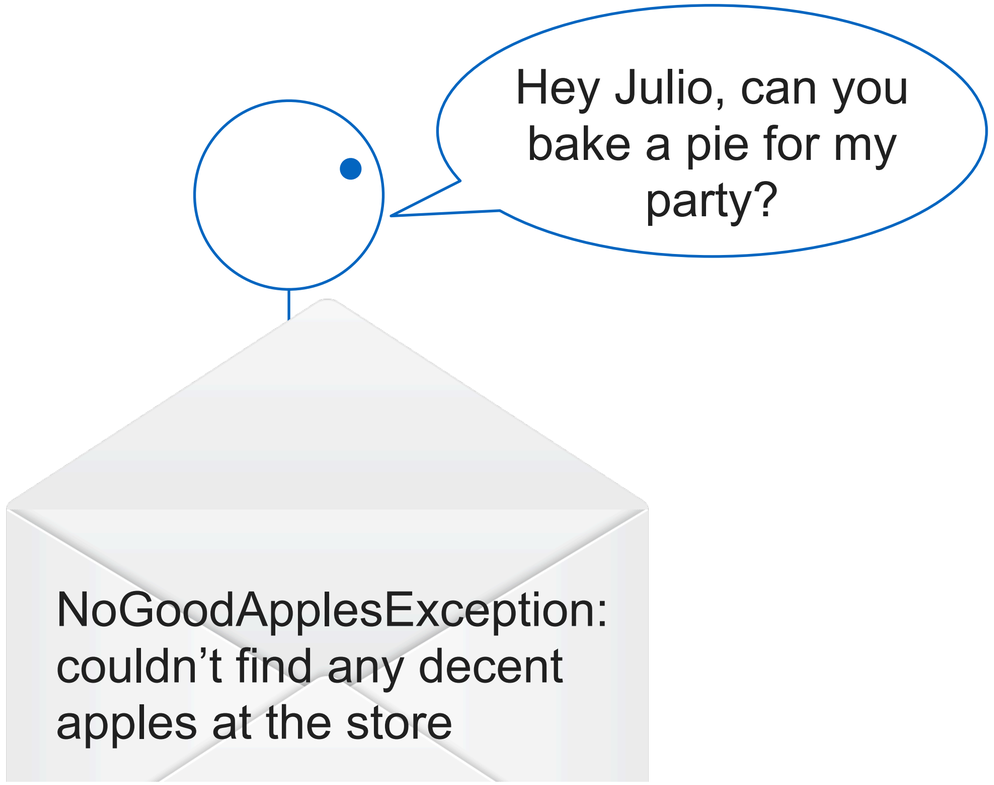
julio:

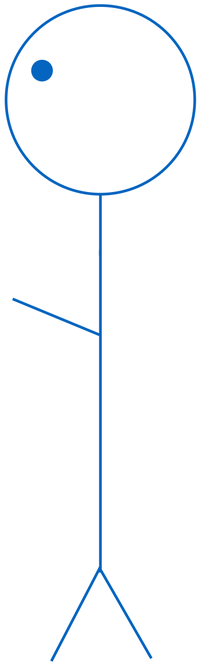
```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

Error handling in C++: Exceptions





int main()

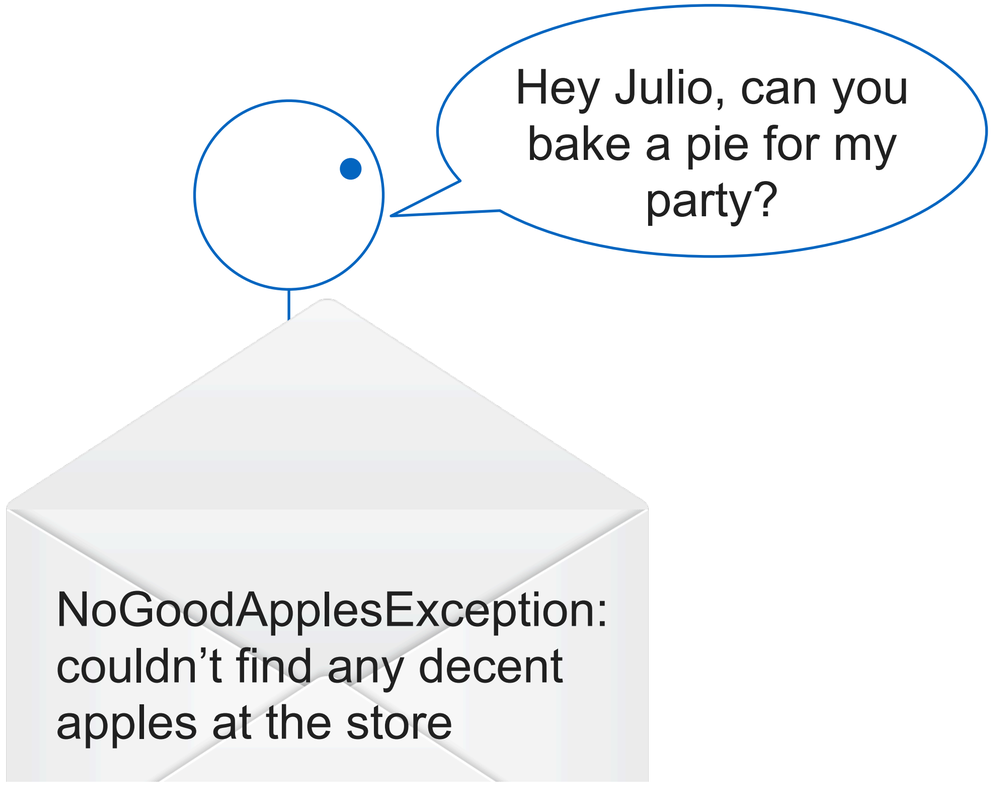
julio:

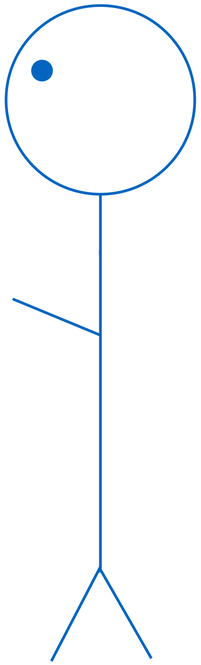
```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

Error handling in C++: Exceptions





```
int main() {
   try {
      make_pie();
   } catch(NoGoodApplesException &e) {
      party_without_pie();
}
```

julio:

```
struct apple_pie *make_pie() {
    get_apples();
    bake_ingredients();
}
```

ryan:

```
struct apple *get_apples()
```

Massive improvements over C-style error handling

- You don’t have to write error propagation code *every* time you call a function that might produce an error
  - Exceptions propagate up the stack automatically until they are handled by a try/catch
- Errors will not go unnoticed
  - Worst case scenario, they’ll propagate up to main() and crash the program
  - Sounds bad, but a crash is much better than the program continuing to run in an undefined state

Except Exceptions

- Why might exceptions not be so hot?
  - Failure modes become hard to reason about: any function can throw any exception at any time
    - Code might fail because of an exception that was thrown by a totally unrelated function twelve function calls away
    - Even harder to manage in evolving codebases as new errors are added
    - Hard to spot where errors may occur
      - What if you call a helper function in a destructor that ends up throwing an exception?
  - Can cause resource leaks and other unexpected behavior
    - Exceptions are forbidden in many codebases for this reason

Exceptions without RAII: sad times

RAII = “resource acquisition is initialization”. In a language with RAII, resources are tied to an object; when object is destroyed, resources are freed. (Ex: C++ destructor.)

```
void process_input() {
    char *buf = malloc(128);

    // read input from user:
    fgets(buf, 128, stdin);
    // do more processing on input:
    some_helper(input);

    free(buf);
}
```

Looks good to me?

```
int main() {
    while (true) {
        try {
            process_input();
        } catch (BadInputError) {
            cerr << "That wasn't valid, try again" << endl;
        }
    }
}
```

```
void some_helper(string input) {
    if (input == "uh oh") {
        throw BadInputError("I don't like that");
    }
}
```

Exceptions without RAII: sad times

RAII = “resource acquisition is initialization”. In a language with RAII, resources are tied to an object; when object is destroyed, resources are freed. (Ex: C++ destructor.)

```
void process_input() {
    char *buf = malloc(128);

    // read input from user:
    fgets(buf, 128, stdin);
    // do more processing on input:
    some_helper(input);

    free(buf);
}
```

```
int main() {
    while (true) {
        try {
            process_input();
        } catch (BadInputError) {
            cerr << "That wasn't valid, try again" << endl;
        }
    }
}
```

```
void some_helper(string input) {
    if (input == "uh oh") {
        throw BadInputError("I don't like that");
    }
}
```

Exceptions without RAII: sad times

RAII = “resource acquisition is initialization”. In a language with RAII, resources are tied to an object; when object is destroyed, resources are freed. (Ex: C++ destructor.)

```
void process
    char *bu

    // read
    fgets(bu
    // do mo
    some_hel

    free(buf
}
```

```
ndl;
```

Looks goo

Video link: https://twitter.com/c0dehard/status/1327718161848872960

Exceptions without RAII: sad times

RAII = “resource acquisition is initialization”. In a language with RAII, resources are tied to an object; when object is destroyed, resources are freed. (Ex: C++ destructor.)

```
void process_input() {
    char *buf = malloc(128);

    // read input from user:
    fgets(buf, 128, stdin);
    // do more processing on input:
    some_helper(input);

    free(buf);
}
```

```
int main() {
    while (true) {
        try {
            process_input();
        } catch (BadInputError) {
            cerr << "That wasn't valid, try again" << endl;
        }
    }
}
```

```
void some_helper(string input) {
    if (input == "uh oh") {
        throw BadInputError("I don't like that");
    }
}
```

MEMORY LEAK!!!!

Error handling in Rust: Enums

Towards better error handling: Enums

- An *enum* (enumeration) is a type that can contain one of several *variants*

```
enum TrafficLightColor {
    Red,
    Yellow,
    Green,
}

let current_state: TrafficLightColor = TrafficLightColor::Green;
```

Type TrafficLightColor

⬅ Variants

- Rust: *match* expression is like a switch statement in C/C++/Java, except all possible variants must be covered

```
fn drive(light_state: TrafficLightColor) {
    match light_state {
        TrafficLightColor::Green => println!("zoom zoom!"),
        TrafficLightColor::Red =>
            println!("sitting like a boulder!"),
    }
}
```

  - The compiler will warn you if there's a possibility you missed!

```
error[E0004]: non-exhaustive patterns: `Yellow` not covered
 --> src/lib.rs:8:11
  |
1 | / enum TrafficLightColor {
2 | |     Red,
3 | |     Yellow,
  | |     ------ not covered
4 | |     Green,
5 | | }
  | |_- `TrafficLightColor` defined here
...
8 |         match light_state {
  |               ^^^^^^^^^^^ pattern `Yellow` not covered
  |
  = help: ensure that all possible cases are being handled, possibly by
adding wildcards or more match arms
  = note: the matched value is of type `TrafficLightColor`
```

Towards better error handling: Enums

- Can use a default binding to catch all other cases if there's only a few you're interested in:

```
match light_state {
    TrafficLightColor::Green => println!("zoom zoom!"),
    _ => println!("do not pass go"),
}
```

⬅ Default binding.

"Do this in all other cases"

Towards better error handling: Enums

- Unlike enums in most common languages, Rust enums can store arbitrary data!

```
enum Location {
    Coordinates(f32, f32),
    Address(String),
    Unknown,
}
```

Example: want to store location of something, & want options for how to represent it:

- Lat/long coords (—> store value as pair of 32-bit floats)
- Address (—> store value as a string)
- Location unknown (—> no data associated)

Towards better error handling: Enums

```
enum Location {
    Coordinates(f32, f32),
    Address(String),
    Unknown,
}
```

You can extract data from variants using a **match** expression:

```
fn print_location(loc: Location) {
    match loc {
        Location::Coordinates(lat, long) => {
            println!("Person is at ({}, {})", lat, long);
        },
        Location::Address(addr) => {
            println!("Person is at {}", addr);
        },
        Location::Unknown => println!("Location unknown!”),
    }
}

print_location(Location::Address(“353 Jane Stanford Way”.to_string()));
```

EnumName::EnumVariant(value_to_store)

Error handling in Rust

- What if we use enums to clearly represent successful returns / errors?
  - If the functions run successfully, return Ok(whatever return value)
  - If an error happens, return Err(some error object)
-
```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

(this Result type is part of the Rust standard library, no need to define it yourself)

We'll talk more about the <T, E> syntax next week. At a high level:

- *T = type of value we want to be stored in Ok on success*
  - *Ex: return an unsigned integer on success*
- *E = type we want to be stored in Err on error*
  - *Ex: return a string with an error message*

Usage of Result

Success return type Error return type

```
fn gen_num_sometimes() -> Result<u32, &'static str> {
    if get_random_num() > 10 {
        Ok(get_random_num()) // returns Ok with a number stored
    } else {
        Err("Spontaneous failure!”) // returns Err with string stored
    }
}

fn main() {
    match gen_num_sometimes() {
        Ok(num) => println!("Got number: {}", num),
        Err(message) => println!(“Operation failed: {}", message),
    }
}
```

Questions?

Comparison to C errors

- We had two main issues with C error handling:
  - It’s too easy to miss errors
  - Proper error handling is too verbose (need too much extra code to propagate errors)
- This fixes the first problem: it’s now obvious from the function signature which functions can return errors, and (because of enum rules) the compiler will verify that you do something with a returned error
- Second problem is still an issue!

Comparison to C errors

- Error handling is still too verbose (with what we have so far):

```
fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();

    let result = File::open(filename);

    let mut f = match result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

io::Error is a type of error meant for when you're doing `io` (input/output) operations, like reading in from a file.

Adapted from https://doc.rust-lang.org/edition-guide/rust-2018/error-handling-and-panics/the-question-mark-operator-for-easier-error-handling.html

Meet the ? operator

- Suppose we have helper_function() -> Result<T, E>
- `let val: T = helper_function()?` means:
  - If `helper_function` returns `Ok(some value)`, set `val` = that value
  - If helper function returns `Err(some error)`, stop and return/propagate that error

```
fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();

    let mut f = match File::open(filename) {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => return Err(e),
    }
}
```

```
fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();

    let mut f = File::open(filename)?;
    // if file was successfully opened, store file in `f`.
    // if an error occurred, stop and return error to caller


    f.read_to_string(&mut s)?;
    Ok(s)
}
```

Meet the ? operator

- Suppose we have helper_function() -> Result<T, E>
- `let val: T = helper_function()?` means:
  - If `helper_function` returns `Ok(some value)`, set `val` = that value
  - If helper function returns `Err(some error)`, stop and return/propagate that error

```rust
fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();

    let mut f = match File::open(filename) {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => return Err(e),
    }
}
```

Even shorter:

```rust
fn read_file(filename: &str) -> Result<String, io::Error> {
    let mut s = String::new();
    File::open(filename)?.read_to_string(&mut s)?;
    Ok(s)
}
```

Meet the ? operator

- Why doesn’t this code compile?

```
fn read_file(filename: &str) -> String {
    let mut contents = String::new();
    File::open(filename)?.read_to_string(&mut contents)?;
    contents
}
```

- Note that the ? operator is for *propagating* errors, and this function returns `String` (i.e. it cannot return an error)

Panics

- What about errors that we don’t wish to propagate/handle?
  - Could be a serious, unrecoverable error
  - Could be an error that we don’t anticipate ever happening and don’t want to put the effort into handling
  - Ex: if we are a terminal program and we fail to read input from the terminal, there isn’t really anything graceful to do
- The `panic!` macro crashes a program immediately with an error message

```
if sad_times() {
  panic!("Sad times!");
}
```

- `Result::unwrap()` and `Result::expect()` allow us to extract the returned value from an `Ok()` result, panicking if we got an `Err`

unwrap() and expect()

```
// File::open returns Result: Ok(file) or Err(error)
// Unwrap means:
// - “if result is Ok: store value inside enum in `file`
// - “if result is Err (opening file failed): panic (crash program)”
// Panic if opening a file fails:
let mut file = File::open(filename).unwrap();
// `expect` is the same as `unwrap`, but allows you to print a
// more descriptive error message when panicking.
let mut file = File::open(filename).expect("Failed to open file");

// One more example with `expect` – panic with a helpful error message
// if reading from standard input fails. (Nothing to return here.)
let mut input = String::new();
io::stdin().read_to_string(&mut input).expect("Failed to read from stdin”);
```

Handling nulls

*“**I call it my billion-dollar mistake.** It was **the invention of the null reference** in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”*

- Tony Hoare

CVE List▾ CNAs▾ WGs▾ Board▾ About▾ News & Blog▾

Common Vulnerabilities and Exposures

NVD
Go to for:
CVSS Scores
CPE Info
Advanced Search

Search CVE List | Download CVE | Data Feeds | Request CVE IDs | Update a CVE Entry

TOTAL CVE Entries: 133847

HOME > CVE > SEARCH RESULTS

Search Results

There are **1627** CVE entries that match your search.

Name	Description
CVE-2020-9759	An issue was discovered in WeeChat before 2.7.1 (0.4.0 to 2.7 are affected). A malformed message 352 (who) can cause a NULL pointer dereference in the callback function, resulting in a crash.
CVE-2020-9385	A NULL Pointer Dereference exists in libzint in Zint 2.7.1 because multiple + characters are mishandled in add_on in upcean.c, when called from eanx in upcean.c during EAN barcode generation.
CVE-2020-9327	In SQLite 3.31.1, isAuxiliaryVtabOperator allows attackers to trigger a NULL pointer dereference and segmentation fault because of generated column optimizations.
CVE-2020-8859	This vulnerability allows remote attackers to create a denial-of-service condition on affected installations of ELOG Electronic Logbook 3.1.4-283534d. Authentication is not required to exploit this vulnerability. The specific flaw exists within the processing of HTTP parameters. A crafted request can trigger the dereference of a null pointer. An attacker can leverage this vulnerability to create a denial-of-service condition. Was ZDI-CAN-10115.
CVE-2020-8448	In OSSEC-HIDS 2.7 through 3.5.0, the server component responsible for log analysis (ossec-analysisd) is vulnerable to a denial of service (NULL pointer dereference) via crafted messages written directly to the analysisd UNIX domain socket by a local user.
CVE-2020-8011	CA Unified Infrastructure Management (Nimsoft/UIM) 9.20 and below contains a null pointer dereference vulnerability in the robot (controller) component. A remote attacker can crash the Controller service.
CVE-2020-8002	A NULL pointer dereference in vrend_renderer.c in virglrenderer through 0.8.1 allows attackers to cause a denial of service via commands that attempt to launch a grid without previously providing a Compute Shader (CS).
CVE-2020-7105	async.c and dict.c in libhiredis.a in hiredis through 0.14.0 allow a NULL pointer dereference because malloc return values are unchecked.
CVE-2020-7062	In PHP versions 7.2.x below 7.2.28, 7.3.x below 7.3.15 and 7.4.x below 7.4.3, when using file upload functionality, if upload progress tracking is enabled, but session.upload_progress.cleanup is set to 0 (disabled), and the file upload fails, the upload procedure would try to clean up data that does not exist and encounter null pointer dereference, which would likely lead to a crash.
CVE-2020-6795	When processing a message that contains multiple S/MIME signatures, a bug in the MIME processing code caused a null pointer dereference, leading to an unexploitable crash. This vulnerability affects Thunderbird < 68.5.
CVE-2020-6631	An issue was discovered in GPAC version 0.8.0. There is a NULL pointer dereference in the function gf_m2ts_stream_process_pmt() in media_tools/m2ts_mux.c.

NULL pointer dereferences

Nulls -> not null damage

- Most null pointer dereferences simply cause crashes (denial of service)… but not all
- CVE-2009-2694 in Pidgin messenger: https://www.cvedetails.com/cve/CVE-2009-2694/
- `msn_slplink_message_find()` retrieves previously-received parts of a message
- Special types of messages (“acknowledgement messages”) don’t have any message contents. `message->buffer` is set to NULL
- When trying to re-assemble received data, `msn_slplink_process_msg()` calls `msn_slplink_message_find()` and then runs `memcpy(slpmsg->buffer + offset, data, len);`
- `slpmsg->buffer` is null, so the attacker-supplied offset can be used to control what memory gets overwritten
- Similar vulnerability in Adobe Acrobat Pro: https://www.zerodayinitiative.com/advisories/ZDI-19-871/

A sticky situation

- See first example!
- Why are NULLs so dangerous?
  - They place a huge burden on the programmer: any time you have a pointer, you need to think, *is it possible for this to be NULL?*
  - Static analyzers can’t warn about all the possible NULLs without being riddled with false positives
- What should we do about it?

```
enum Option<T> {
    None,
    Some(T),
}
```

(also from in the standard library)

(very similar to Optional in modern C++)

(Unwrap, Expect, and `?` work here too!)

```rust
fn feeling_lucky() -> Option<String> {
    if get_random_num() > 10 {
        Some(String::from("I'm feeling lucky!"))
    } else {
        None
    }
}
```

```rust
fn feeling_lucky() -> Option<String> {
    if get_random_num() > 10 {
        Some(String::from("I'm feeling lucky!"))
    } else {
        None
    }
}
```

```rust
match feeling_lucky() {
    Some(message) => {
        println!("Got message: {}", message);
    },
    None => {
        println!("No message returned :-/");
    },
}
```

```rust
fn feeling_lucky() -> Option<String> {
    if get_random_num() > 10 {
        Some(String::from("I'm feeling lucky!"))
    } else {
        None
    }
}


// Check if is_none/is_some():
if feeling_lucky().is_none() {
    println!("Not feeling lucky :(");
}

// unwrap/expect work here too:
let message = feeling_lucky().unwrap();
let message = feeling_lucky().expect("feeling_lucky failed us!");

// you can also provide a default value in case None was returned:
let message = feeling_lucky().unwrap_or("Not lucky :(“.to_string());

// ? operator also works in functions that return Option:
let expanded_message: String = feeling_lucky()? + " Are you?";
```