Rust: making the Internet more secure, statically

Keith Wansbrough, Metaswitch Networks Ltd.

2018-01-24

metaswitch

Outline

- Common vulnerabilities in software today
- The Rust programming language
- How Rust's type system ensures memory safety:
  - Ownership / Linear types
  - Lifetimes / Region types
- API safety – more than just pointers
- High performance communications software
- Conclusion and further reading

metaswitch

Common vulnerabilities

metaswitch

Some recent vulnerabilities in Apache httpd

Apache httpd 2.2 vulnerabilities

This page lists all security vulnerabilities fixed in released versions of Apache httpd 2.2. Each vulnerability is given a security impact rating by the Apache security team - please note that this rating may we
Apache httpd the flaw is known to affect, and where a flaw has not been verified list the version with a question mark.

Please note that if a vulnerability is shown below as being fixed in a "-dev" release then this means that a fix has been applied to the development source tree and will be part of an upcoming full release.

This page is created from a database of vulnerabilities originally populated by Apache Week. Please send comments or corrections for these vulnerabilities to the Security Team.

Fixed in Apache httpd 2.2.35-dev

low: Use-after-free when using <Limit > with an unrecognized method in .htaccess ("OptionsBleed") (CVE-2017-9798)

When an unrecognized HTTP Method is given in an <Limit {method}> directive in an .htaccess file, and that .htaccess file is processed by the corresponding request, the global methods table is corrupted in the current worker process, resulting in erratic behaviour.

This behavior may be avoided by listing all unusual HTTP Methods in a global httpd.conf RegisterHttpMethod directive in httpd release 2.2.32 and later.

To permit other .htaccess directives while denying the <Limit > directive, see the AllowOverrideList directive.

Source code patch is at;

- http://www.apache.org/dist/httpd/patches/apply_to_2.2.34/CVE-2017-9798-patch-2.2.patch

Note 2.2 is end-of-life, no further release with this fix is planned. Users are encouraged to migrate to 2.4.28 or later for this and other fixes.

Acknowledgements: We would like to thank Hanno Böck for reporting this issue.

Reported to security team	12th July 2017
Issue public	18th September 2017
Affects	2.2.34, 2.2.32, 2.2.31, 2.2.29, 2.2.27, 2.2.26, 2.2.25, 2.2.24, 2.2.23, 2.2.22, 2.2.21, 2.2.20, 2.2.19, 2.2.18, 2.2.17, 2.2.16, 2.2.15, 2.2.14, 2.2.13, 2.2.12, 2.2.11, 2.2.10, 2.2.9, 2.2.8, 2.2.6, 2.2.5, 2.2.4, 2.2.3, 2.2.2, 2.2.0

Fixed in Apache httpd 2.2.34

important: Uninitialized memory reflection in mod_auth_digest (CVE-2017-9788)

The value placeholder in [Proxy-]Authorization headers of type 'Digest' was not initialized or reset before or between successive key=value assignments. by mod_auth_digest.

Providing an initial key with no '=' assignment could reflect the stale value of uninitialized pool memory used by the prior request, leading to leakage of potentially confidential information, and a segfault.

Acknowledgements: We would like to thank Robert Święcki for reporting this issue.

gi?name=CVE-2017-9798

metaswitch

Some recent vulnerabilities in Apache httpd

APACHE HTTP SERVER PROJECT

Apache httpd 2.2 vulnerabilities

Fixed in Apache httpd 2.2.35-dev

low: Use-after-free when using <Limit > with an unrecognized method in .htaccess ("OptionsBleed")

Fixed in Apache httpd 2.2.34

Important: Uninitialized memory reflection in mod_auth_digest (CVE-2017-9788)

Important: ap_get_basic_auth_pw() Authentication Bypass (CVE-2017-3167)

Important: mod_ssl Null Pointer Dereference (CVE-2017-3169)

Important: ap_find_token() Buffer Overread (CVE-2017-7668)

Important: mod_mime Buffer Overread (CVE-2017-7679)

Fixed in Apache httpd 2.2.32

Important: Apache HTTP Request Parsing Whitespace Defects (CVE-2016-8743)

n/a: HTTP_PROXY environment variable "httpoxy" mitigation (CVE-2016-5387)

Fixed in Apache httpd 2.2.31

low: HTTP request smuggling attack against chunked request parser (CVE-2015-3183)

Fixed in Apache httpd 2.2.29

Important: mod_cgid denial of service (CVE-2014-0231)

low: HTTP Trailers processing bypass (CVE-2013-5704)

moderate: mod_deflate denial of service (CVE-2014-0118)

moderate: mod_status buffer overflow (CVE-2014-0226)

Fixed in Apache httpd 2.2.27

low: mod_log_config crash (CVE-2014-0098)

moderate: mod_dav crash (CVE-2013-6438)

Fixed in Apache httpd 2.2.25

low: mod_rewrite log escape filtering (CVE-2013-1862)

moderate: mod_dav crash (CVE-2013-1896)

Fixed in Apache httpd 2.2.24

low: XSS due to unescaped hostnames (CVE-2012-3499)

moderate: XSS in mod_proxy_balancer (CVE-2012-4558)

Fixed in Apache httpd 2.2.23

low: XSS in mod_negotiation when untrusted uploads are supported (CVE-2012-2687)

…

http://httpd.apache.org/security/vulnerabilities_22.html

metaswitch

CVE-2010-0425 (found by senseofsecurity.com.au)

- CVE: “modules/arch/win32/mod_isapi.c in mod_isapi in the Apache HTTP Server 2.0.37 through 2.0.63, 2.2.0 through 2.2.14, and 2.3.x before 2.3.7, when running on Windows, does not ensure that request processing is complete before calling isapi_unload for an ISAPI .dll module, which allows remote attackers to execute arbitrary code via unspecified vectors related to a crafted request, a reset packet, and "orphaned callback pointers."”
- Initial report: “By sending a specially crafted request followed by a reset packet it is possible to trigger a vulnerability in Apache mod_isapi that will unload the target ISAPI module from memory. However function pointers still remain in memory and are called when published ISAPI functions are referenced. This results in a dangling pointer vulnerability. Successful exploitation results in the execution of arbitrary code with SYSTEM privileges.”

Common Vulnerabilities and Exposures
The Standard for Information Security Vulnerability Names

Dangling pointer vulnerability in Apache httpd (CVE-2010-0425)

- Load a module to handle a request.
- Install a callback to invoke the module.
- Request fails.
- Unload the module.
- Trigger the callback (e.g., by a subsequent request which reloads the module).
- Callback tries to call the module at the old address,
  - but instead calls the new contents of that address – which are now part of the second request.
- Request has been carefully crafted to put x86 instructions at that location.
- Attacker’s code is executed.

metaswitch

The fix

```
Do not unload an isapi .dll module until the request processing is completed, avoiding orphaned callback pointers.
Submitted by: Brett Gervasoni <brettg senseofsecurity.com>, trawick
Reviewed by: trawick, wrowe

-- httpd/httpd/trunk/modules/arch/win32/mod_isapi.c     2010/03/02 04:18:45         917869
+++ httpd/httpd/trunk/modules/arch/win32/mod_isapi.c    2010/03/02 04:30:33         917870
@@ -1503,7 +1503,6 @@
     /* Set up the callbacks */
     cid->ecb->ReadClient = ReadClient; … // for example; others omitted for space on slide

     /* Set up client input */
     res = ap_setup_client_block(r, REQUEST_CHUNKED_ERROR);
     if (res) {
-        isapi_unload(isa, 0);
         return res;
     }

@@ -1534,7 +1533,6 @@
         }

         if (res < 0) {
-            isapi_unload(isa, 0);
             return HTTP_INTERNAL_SERVER_ERROR;
         }
```

- http://svn.apache.org/viewvc?view=revision&revision=917870

metaswitch

The fix

```
Do not unload an isapi .dll module until the request process
Submitted by: Brett Gervasoni <brettg senseofsecurity.com>,
Reviewed by: trawick, wrowe

-- httpd/httpd/trunk/modules/arch/win32/mod_isapi.c     201
+++ httpd/httpd/trunk/modules/arch/win32/mod_isapi.c    201
@@ -1503,7 +1503,6 @@
     /* Set up the callbacks */
     cid->ecb->ReadClient = ReadClient; … // for example; others omitted for space on slide

     /* Set up client input */
     res = ap_setup_client_block(r, REQUEST_CHUNKED_ERROR);
     if (res) {
-        isapi_unload(isa, 0);
         return res;
     }

@@ -1534,7 +1533,6 @@
         }

         if (res < 0) {
-            isapi_unload(isa, 0);
             return HTTP_INTERNAL_SERVER_ERROR;
         }
```

- Need two rules to fix:
  - Once something is freed, it can’t be used (***ownership***).
  - An object must be valid as long as any reference to it exists (***lifetime***).

- http://svn.apache.org/viewvc?view=revision&revision=917870

metaswitch

Rust

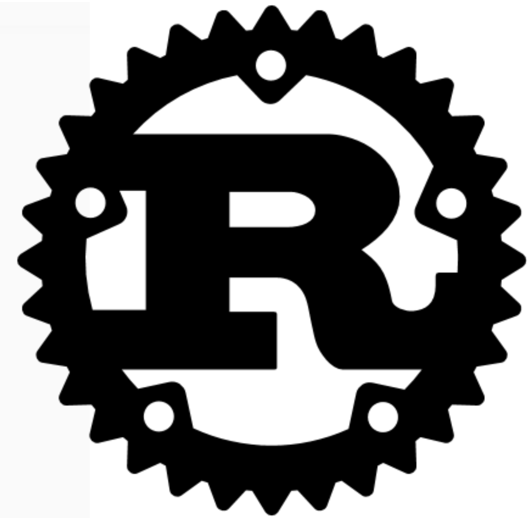
metaswitch

The Rust programming language

- A modern programming language (2009), originated and sponsored by Mozilla Labs (makers of Firefox)
- Focus on performance and safety
  - *“Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.”* – [rust-lang.org]()
  - *“Rust is a systems language pursuing the trifecta: safe, concurrent, and fast”* – [*This Week in Rust*]()

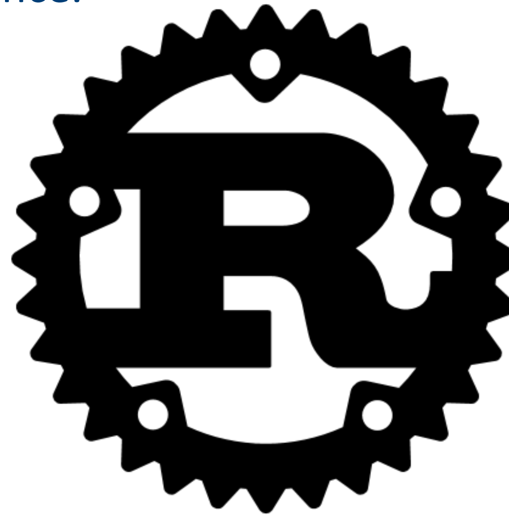
```
fn main() {
    let greetings = ["Hello", "Ciao", "こんにちは",
                     "안녕하세요", "Cześć", "Olá"];

    for (num, greeting) in greetings.iter().enumerate() {
        print!("{} : ", greeting);
        match num {
            0 =>  println!("This code is editable and runnable!"),
            1 =>  println!("Questo codice è modificabile ed eseguibile!"),
            2 =>  println!("このコードは編集して実行出来ます！"),
            3 =>  println!("여기에서 코드를 수정하고 실행할 수 있습니다!"),
            4 =>  println!("Ten kod można edytować oraz uruchomić!"),
            5 =>  println!("Este código é editável e executável!"),
            _ =>  {},
        }
    }
}
```



Fast, concurrent, safe

- ***Fast***
  - Compiles to native machine code. Competes with C/C++ for raw speed.
  - No runtime, no scheduler, no startup.
  - No garbage collector – predictable performance.
  - Zero-cost abstractions.
- ***Concurrent***
  - Multi-threaded or coroutine-based
  - Can use multiple cores
  - Protection from data races
  - Threadsafe toolkit in standard library
- ***Safe***
  - Memory safety: no double frees, no NULL pointers, no dangling pointers, no scribblers
  - API safety: compile-time API checking, no "unexpected behaviour"
- *It's also pleasant to use: modern features, good IDEs, clear error messages, etc.*



Other languages:

C/C++:
- Fast: yes
- Concurrent: yes
- Safe: no

Java/Scala:
- Fast: sort-of
- Concurrent: yes
- Safe: sort-of

metaswitch

Ownership in Rust

```
fn foo() {
    let v = vec![1, 2, 3];
}
```


metaswitch

Ownership in Rust

```
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] is: {}", v[0]);
```


metaswitch

Ownership in Rust

```rust
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] is: {}", v[0]);
```

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```


metaswitch

Ownership in Rust

```rust
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] is: {}", v[0]);
```

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```

```rust
fn take(v: Vec<i32>) {
    // What happens here isn't important.
}

let v = vec![1, 2, 3];

take(v);

println!("v[0] is: {}", v[0]);
```


metaswitch

Ownership in Rust

```rust
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] is: {}", v[0]);
```

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```

```rust
fn take(v: Vec<i32>) {
    // What happens here isn’t important.
}

let v = vec![1, 2, 3];

take(v);

println!("v[0] is: {}", v[0]);
```

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```


metaswitch

Borrowing

```
fn main() {
    // This borrows an immutable reference.
    fn sum_vec(v: &Vec<i32>) -> i32 {
        v.iter().fold(0, |a, &b| a + b)
    }
    // Borrow two vectors and sum them.
    // This kind of borrowing does not allow mutation through the borrowed reference.
    fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
        // Do stuff with `v1` and `v2`.
        let s1 = sum_vec(v1);
        let s2 = sum_vec(v2);
        // Return the answer.
        s1 + s2
    }

    let v1 = vec![1, 2, 3];
    let v2 = vec![4, 5, 6];

    let answer = foo(&v1, &v2);
    println!("{}", answer);
}
```


metaswitch

Borrowing

```
fn main() {
    // This borrows an immutable reference.
    fn sum_vec(v: &Vec<i32>) -> i32 {
        v.iter().fold(0, |a, &b| a + b)
    }
    // Borrow two vectors and sum them.
    // This kind of borrowing does not allow mutation through the borrowed reference.
    fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
        // Do stuff with `v1` and `v2`.
        let s1 = sum_vec(v1);
        let s2 = sum_vec(v2);
        // Return the answer.
        s1 + s2
    }

    let v1 = vec![1, 2, 3];
    let v2 = vec![4, 5, 6];

    let answer = foo(&v1, &v2);
    println!("{}", answer);
}
```

Can't mutate via v1 or v2.

E.g., `v1.push(42);` yields:

```
error: cannot borrow
immutable borrowed content `*v1`
as mutable
```


metaswitch

Mutable references

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```


metaswitch

Mutable references

```rust
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

```
6
```


metaswitch

Mutable references

```
let mut x = 5;

let y = &mut x;
*y += 1;

println!("{}", x);
```


metaswitch

Mutable references

```
let mut x = 5;

let y = &mut x;
*y += 1;

println!("{}", x);
```

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
note: previous borrow of `x` occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of `x` until the borrow ends
        let y = &mut x;
                     ^
note: previous borrow ends here
fn main() {

}
^
```


metaswitch

Borrowing rules in Rust

- Any borrow must last for a scope no greater than that of the owner.
- Each resource may have **either one or the other** of these two kinds of borrows, but not both at the same time:
  - One or more references (&T) to a resource.
  - Exactly one mutable reference (&mut T) to a resource.

- These rules are sufficient to make data races impossible in Rust!

> There is a 'data race' when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized.

- Also prevents things like:
  - Iterator invalidation (mutating a collection while you're iterating over it).
  - Use after free

metaswitch

Linear types = ownership

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma_1 \vdash e_1 : T_2 \to T_1 \qquad \Gamma_2 \vdash e_2 : T_2}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : T_1} \text{(App)}$$

Girard, Linear Logic, 1987
Wadler, A taste of linear logic, 1993
Jung et al, RustBelt, 2018

metaswitch

Linear types = ownership

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma_1 \vdash e_1 : T_2 \rightarrow T_1 \qquad \Gamma_2 \vdash e_2 : T_2}{\Gamma_1, \Gamma_2 \vdash e_1 \, e_2 : T_1} \text{ (App)}$$

$$\cfrac{\cfrac{\overline{cake : T \vdash cake : T}}{cake : T \vdash have(cake) : ()} \qquad \cfrac{\overline{bread : T \vdash bread : T}}{bread : T \vdash eat(bread) : ()}}{cake : T,\ bread : T \vdash have(cake);\ eat(bread) : ()}$$

Girard, Linear Logic, 1987
Wadler, A taste of linear logic, 1993
Jung et al, RustBelt, 2018

metaswitch

Linear types = ownership

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma_1 \vdash e_1 : T_2 \to T_1 \qquad \Gamma_2 \vdash e_2 : T_2}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : T_1} \text{(App)}$$

$$\dfrac{\dfrac{\overline{cake : T \vdash cake : T}}{cake : T \vdash have(cake) : ()} \qquad \dfrac{\overline{bread : T \vdash cake : ?}}{bread : T \vdash eat(cake) : ?}}{cake : T,\ bread : T \vdash have(cake);\ eat(cake) : ?}$$

Girard, Linear Logic, 1987
Wadler, A taste of linear logic, 1993
Jung et al, RustBelt, 2018

metaswitch

Linear types = ownership

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma_1 \vdash e_1 : T_2 \rightarrow T_1 \qquad \Gamma_2 \vdash e_2 : T_2}{\Gamma_1, \Gamma_2 \vdash e_1 e_2 : T_1} \text{(App)}$$

$$\frac{\dfrac{\overline{cake : T \vdash cake : T}}{cake : T \vdash have(cake) : ()} \qquad \dfrac{\overline{bread : T \vdash cake : \mathbf{error!}}}{bread : T \vdash eat(cake) : \mathbf{error!}}}{cake : T, bread : T \vdash have(cake); eat(cake) : \mathbf{error!}}$$

Girard, Linear Logic, 1987
Wadler, A taste of linear logic, 1993
Jung et al, RustBelt, 2018

metaswitch

Lifetimes

metaswitch

Lifetimes in Rust

- Lending out a reference to a resource owned by someone else:
  - I acquire a handle to some kind of resource.
  - I lend you a reference to the resource.
  - I decide I’m done with the resource, and deallocate it, while you still have your reference.
  - You decide to use the resource. *BOOM – dangling pointer / use after free!*

```
let r;                  // Introduce reference: `r`.
{
    let i = 1;          // Introduce scoped value: `i`.
    r = &i;             // Store reference of `i` in `r`.
}                       // `i` goes out of scope and is dropped.

println!("{}", r);  // `r` still refers to `i`.
```

- (Rust uses explicit memory management – not garbage collection.)

metaswitch

Lifetimes

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    // ...
}

let line = "lang:en=Hello World!";
let lang = "en";

let v;
{
    let p = format!("lang:{}=", lang);  // -+ `p` comes into scope.
    v = skip_prefix(line, p.as_str());  //  |
}                                       // -+ `p` goes out of scope.
println!("{}", v);
```


metaswitch

Lifetimes

```
fn skip_prefix<'a, 'b>(line: &'a str, prefix: &'b str) -> &'a str {
    // ...
}

let line = "lang:en=Hello World!";
let lang = "en";

let v;
{
    let p = format!("lang:{}=", lang);  // -+ `p` comes into scope.
    v = skip_prefix(line, p.as_str());  //  |
}                                       // -+ `p` goes out of scope.
println!("{}", v);
```


metaswitch

More lifetimes

```rust
struct Foo<'a> {
    x: &'a i32,
}

fn show() {
    let y = &5; // This is the same as `let _y = 5; let y = &_y;`.
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

```rust
let x: &'static str = "Hello, world.";
```

- Inference and elision

metaswitch

Region types = lifetimes

```
letregion ρ4,ρ5 in
  let f =
    letregion ρ6 in
      let x = (2 at ρ2, 3 at ρ6) at ρ4;
        (|y| (x.0,y) at ρ1) at ρ5
  at ρ5;
  f(5 at ρ3)
```

- Annotate program and types with regions in which allocation occurs.
- Constrain which expressions can use which regions.

Tofte & Talpin, Region-Based Memory Management, 1994
Morrisett et al, Region-Based Memory Management in Cyclone, 2001
Jung et al, RustBelt, 2018

metaswitch

In practice

- The type system is picky, but it ensures that Rust code cannot access a dangling pointer, or change a value under the feet of another caller.
- These guarantees are zero-cost: all the work is done at compile time.
- Learning curve of using a type system:
  - “Many new users to Rust experience something we like to call ‘fighting with the borrow checker’, where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer’s mental model of how ownership should work doesn’t match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.”

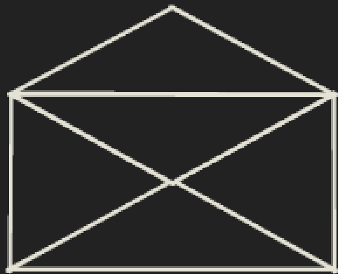
metaswitch

More than just pointers

metaswitch

Resources

```
#[derive(Clone)]
pub struct Letter {
    text: String,
}
```



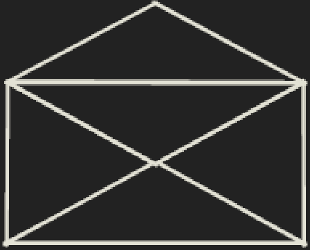
```
pub struct Envelope {
    letter: Option<Letter>,
}
```

```
pub struct PickupLorryHandle {
    done: bool,
    // references to lorry's resources
}
```

From: *A hammer you can only hold by the handle* (video, slides)
Andrea Lattuada (ETH Zürich), RustFest 2017

metaswitch

A dangerous API 1



```
pub struct Envelope {
    letter: Option<Letter>,
}


impl Envelope {
    pub fn wrap(&mut self, letter: &Letter) {
        self.letter = Some(letter.clone());
    }
}

pub fn buy_prestamped_envelope() -> Envelope {
    Envelope { letter: None }
}
```


metaswitch

A dangerous API 2

```rust
pub struct PickupLorryHandle {
    done: bool,
    // references to lorry's resources
}

impl PickupLorryHandle {
    pub fn pickup(&mut self, envelope: &Envelope) {
        /* give letter */
    }
    pub fn done(&mut self) {
        self.done = true; println!("sent");
    }
}

pub fn order_pickup() -> PickupLorryHandle {
    PickupLorryHandle { done: false, /* other handles */ }
}
```


metaswitch

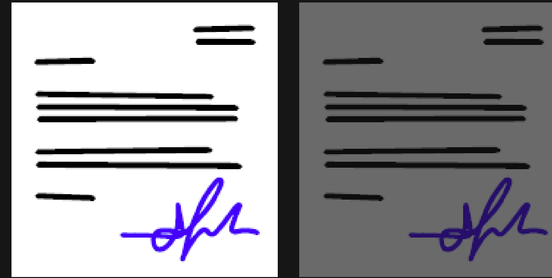
Using the dangerous API

```
// in a separate module
fn main() {
    let rustfest_letter = Letter::new(String::from("Dear RustFest"));
    let mut rustfest_envelope = buy_prestamped_envelope();
    rustfest_envelope.wrap(&rustfest_letter);
    let mut lorry = order_pickup();
    lorry.pickup(&rustfest_envelope);
    lorry.done();
}
```

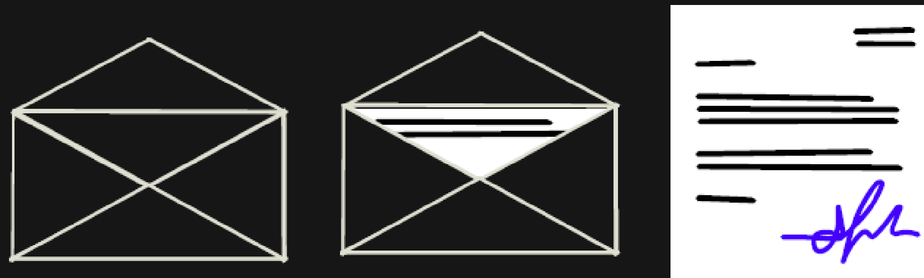

metaswitch

What can go wrong?

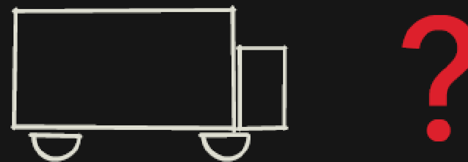
1.



2.



3.



metaswitch

Constraining the dangerous API with ownership types

```
pub struct EmptyEnvelope { }

pub struct ClosedEnvelope {
    letter: Letter,
}
-----------------------------------------------------------------
impl EmptyEnvelope {
    pub fn wrap(self, letter: Letter) -> ClosedEnvelope {
        ClosedEnvelope { letter: letter }
    }
}
-----------------------------------------------------------------
impl PickupLorryHandle {
    pub fn pickup(&mut self, envelope: ClosedEnvelope) {
        /* give letter */
    }
-----------------------------------------------------------------
pub fn buy_prestamped_envelope() -> EmptyEnvelope { EmptyEnvelope { } }
-----------------------------------------------------------------
```

metaswitch

Attempting to put the same letter into two envelopes

```
fn main() {
    let rustfest_letter = Letter::new(String::from("Dear RustFest"));
    let mut envelopes = vec![
        buy_prestamped_envelope(), buy_prestamped_envelope()];
    let mut lorry = order_pickup();
    for e in envelopes.iter_mut() {
        e.wrap(rustfest_letter);

        lorry.pickup(&e);
    }
    lorry.done();
}
```


metaswitch

Attempting to put the same letter into two envelopes

```
error[E0382]: use of moved value:
`rustfest_letter`

fn main() {
    let rustfest_letter = Letter::new(String::from("Dear RustFest"));
    let mut envelopes = vec![
        buy_prestamped_envelope(), buy_prestamped_envelope()];
    let mut lorry = order_pickup();
    for e in envelopes.iter_mut() {
        e.wrap(rustfest_letter);
               ^^^^^^^^^^^^^^^ value moved here in previous iteration
        lorry.pickup(&e);      of loop
    }
    lorry.done();
}
```


metaswitch

High-performance comms software

metaswitch

Metaswitch Networks

- “Metaswitch is the world’s leading cloud native communications software company. We develop commercial and open-source software solutions that are constructively disrupting the way that service providers build, scale, innovate and account for communication services.”

- Write and sell telecoms/networking software
  - Some mobile and web apps…
  - …but mainly back-end servers to make it all work
- Work with customers to put it all together
- Support it

- Complexity, scale, reliability, …

https://upload.wikimedia.org/wikipedia/commons/e/e5/Nanoscience_High-Performance_Computing_Facility.jpg

Metaswitch Networks

- Shift to microservices architecture
- Great opportunity to re-select a modern language to use henceforth
  - Existing code: C, C++, Java, Scala, Python, Perl
    - Productive, but mixed experiences with all of them
  - Needed: safety, flexibility, FFI, tooling, ecosystem, and fun!
    - Rust ticked all the boxes.
- We're well along the path, gaining experience, and loving it.

https://upload.wikimedia.org/wikipedia/commons/e/e5/Nanoscience_High-Performance_Computing_Facility.jpg

Conclusion

metaswitch

Conclusion

- Software engineering has a problem
  - Common vulnerabilities
- PL type systems offer a solution
  - Linear types and region types
- Rust shows that it works in practice
  - Fast, concurrent, safe – pick three
  - Easily adopted
  - Used in production

Further reading:

- Francois Pottier, *A Linear Bestiary*
- Ralf Jung et al, *RustBelt: Securing the Foundations of the Rust Programming Language (POPL 2018)*
- https://www.rust-lang.org/
- Wikipedia on Rust
- Rust Book (1st ed) bibliography - types
- The Rust Book (1st ed): chapter on ownership
- Metaswitch blog (e.g., 1, 2)

Contact me: Keith.Wansbrough@metaswitch.com

@kw217

metaswitch

metaswitch