DISA

Migration to Memory Safe Language: At Speed, At Scale

DISA Technical Exchange Meeting

July 25, 2024

- Ray DeMeo, Chief Growth Officer
- Thomas Wahl, PhD., Principal Researcher
- Jason Rowley, Growth Team

GRAMMATECH

Securing The Software That Secures The World

Agenda

1. The Problem
2. US Government Call to Action
3. Memory Safe in support of DISA Next Strategy
4. Demonstration
5. Q&A

GrammaTech Overview

- **US Based, centered across Central MD, Ithaca NY & FL Space Coast**
  - Highly technical small business in support of DoD & IC
  - World-class solution and research expertise (>50% PhDs)

- **Providing**
  - Advanced Cyber Services, Products, Research, Intelligence, and Mission Support
  - 35 years experience tackling the hardest cybersecurity problems, 151 SBIR/STTRs

- **Value**
  - Software security, resilience, sustainment, automation, and developer productivity

Federal Call to Action

C++ is principally unsafe

June 2023 – **DHS CISA** Joint Guide for S/W Mfgrs.

- Highlights memory safety vulnerabilities as a major security concern and encourages adoption of memory safe languages

December 2023 – The Case for Memory Safe Roadmaps

- Joint statement: **CISA, NSA, FBI, Allied Nations**
- Call to implement roadmaps for migration of critical s/w to memory-safe languages
- Recommends migration to: C#, Go, Java, Python, Rust & Swift

February 2024 – **White House** Press Release

- **Office of the National Cyber Director** report

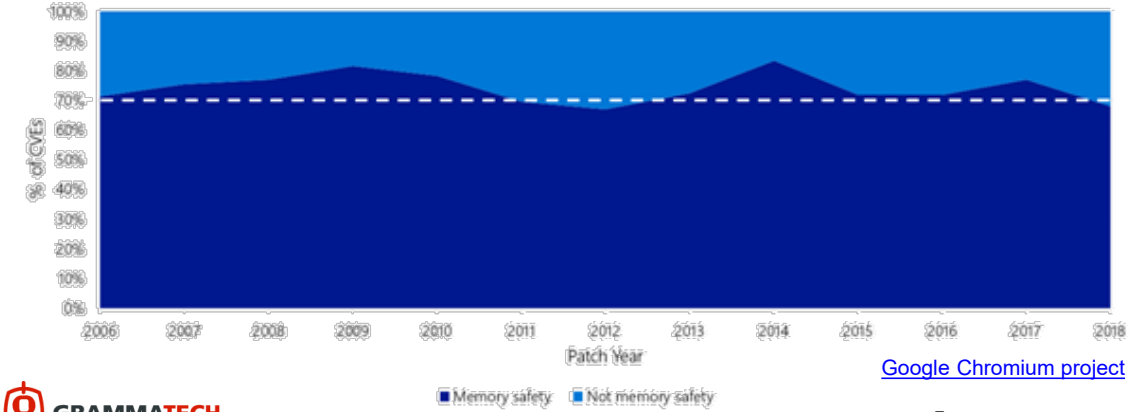
The Problem

DoD is forced to shield networks made of vulnerable components built with C++ code

Yet - 70% of critical vulnerabilities - and their associated costs - can be avoided simply by moving to memory-safe language

Microsoft's Security Response Center (MSRC)

70% is an amazingly high number!



Google Chromium project

Some of the most infamous cyber events in history caused real-world damage

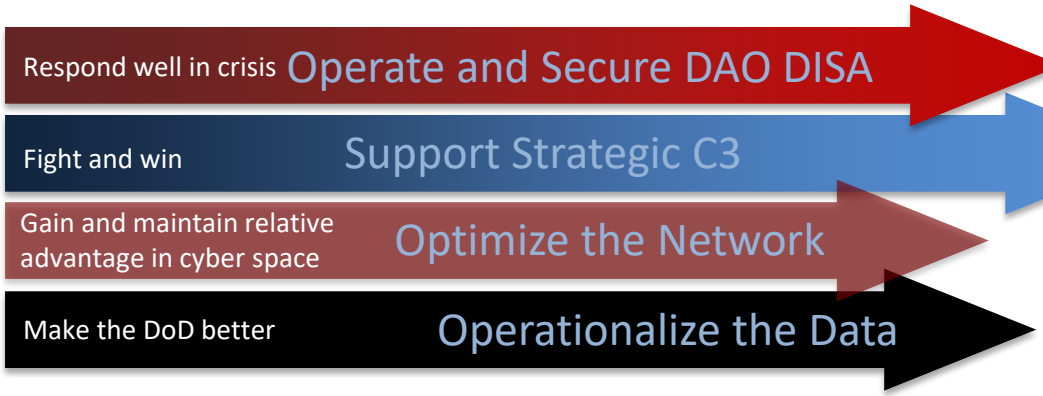
- **2003 Slammer worm**
- **2014 Heartbleed vulnerability**
- **2016 Trident exploit**
- **2023 Blastpass exploit**

The common root cause:
Memory safety vulnerabilities

DHS CISA

July 2024 global outages from CrowdStrike update reported to be a C++ error. Cost estimated to be in excess of $24B

DISA Next Strategy FY 2025-2029



- Defense Information System Network
- Hybrid Cloud Environment
- National Leadership Command Capability
- Joint /Coalition Warfighting Tools
- Consolidated Network
- Zero Trust Tools
- Data Management
- Workforce

All depend upon Memory Safe Architecture

C++ is the Achilles Heel to DODIN

Some notable products on the DoDIN APL include:

Network

- Cisco Catalyst Series Switches
- Cisco ISR (Integrated Services Routers)
- Juniper EX Series Switches
- Juniper SRX Series Services Gateways
- HPE Aruba Switches
- HPE FlexNetwork Routers
- Palo Alto Next-Generation Firewalls
- Fortinet FortiGate Firewalls

Wireless

- Cisco Aironet Series Access Points
- Cisco Meraki Wireless Access Points
- HPE Aruba Wireless Access Points
- HPE Aruba Mobility Controllers
- Juniper Mist Wireless Access Points
- Brocade Ruckus Wireless Access Points

Radio

- Harris Falcon III AN/PRC-152A Wideband Networking Handheld Radio
- Harris Falcon III AN/PRC-117G Multiband Manpack Radio
- Thales AN/PRC-148 JEM (JTRS Enhanced MBITR) Radio
- Thales AN/PRC-154 Rifleman Radio
- Motorola APX Series P25 Two-Way Radios
- Motorola XTS Series Digital Portable Radios

Satellite

- Hughes HX System
- Hughes JUPITER System
- Inmarsat Global Xpress
- Inmarsat L-band services
- Intelsat EpicNG
- IntelsatOne Flex
- SpaceX Starlink
- OneWeb LEO Satellite Services

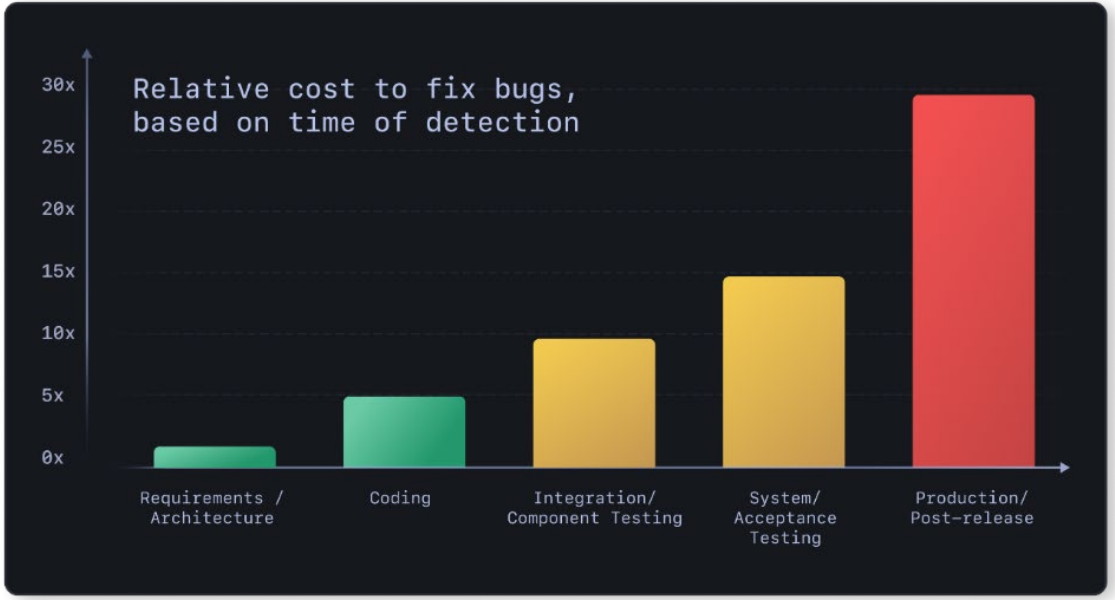
The Cost

Think of designing or deploying anything of consequence…

…knowing that the most substantial failure mechanism is already included in your design when safer alternatives exist

Not a so-called corner-case:

It’s the largest, most obvious point of failure – likely the greatest cost to cyber-protect and maintain that system over its lifetime



C++ to Rust Assisted Migration (CRAM)

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001123C0079. The views, opinions, and/or findings expressed in this document are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Modern, Safe(r) Languages

“Safe language:” memory access via an interface that provably rules out certain errors (use after free, uninitialized read).

Rust:

- **Safe:** enforced via concept of *ownership*
- **Efficient:** no need for garbage collection (unlike C#, Go, Java)
- **Modern:** supportive build system, active community

But what do we do about C++ legacy code?

DARPA: Lifting Legacy Code to Safer Languages

Goal: semi-automatic migration of legacy C/C++ code

Target: (your favorite) *safe* programming language

May: assume well-designed C/C++ code

Must: take advantage of target's idiomatic features

Vision: From C++ to Rust

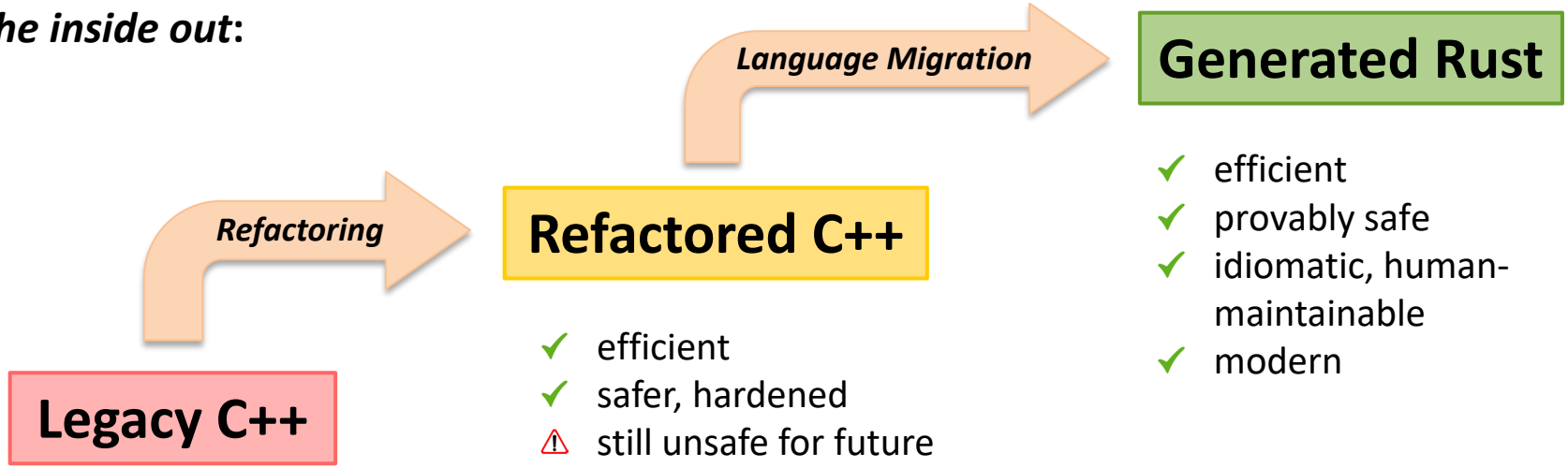
C++ is principally unsafe, yet it does not *force* you to program in a hazardous way.

Put another way:

"Rust is like C++, except the [language semantics] forces programmers to do what they should be doing anyway."

Migration To Memory Safe Languages

Migrating C++ to Rust
***from the inside out*:**



Stages: Refactoring and Migration

Stage-1 Example: `Const` Hardening

C++: variables are (unlike Rust variables) by default *mutable*.

We change them to `const` whenever possible.

- ✓ prevents some semantic programming errors
- ✓ facilitates analysis

refactoring

```
    double segdist = p1->Distance(*p2);
                     ↓
const double segdist = p1->Distance(*p2);
```

Stage-1 Example: Breaking Up Alias Nests

Refactoring multiple non-const references to same memory cell:

```
void multiple_borrows() {

  Point p;
  Point* p0 = &p;
  Point& p1 = *p0;

  f(p1);
  g(p);
}
```

```
void single_borrow() {

  Point p;
  Point* p0 = &p;
  // Point& p1 = *p0;

  f(*p0);
  g(*p0);
}
```

General principle: “Required in Rust, *safer* in C++.”

Stage-2 Example: Container Traversal

```
for (std::vector<Point>::iterator
       p1 = pts.begin(), p2 = std::next(pts.begin());
       p2 != pts.end(); ++p1, ++p2)       { ... }
```

What defines a *container traversal idiom*?

1. **Direction:** left-to-right vs. right-to-left traversal?
2. **Mutation:** is the container content modified?
3. **Indexing:** 1, 2, 3 iterator variables?

CRAM: 1. *abstracts* traversal code to idiom level (using static analysis)
2. *retargets* idiom to Rust (using migration library)

Sample Results

Code Readability

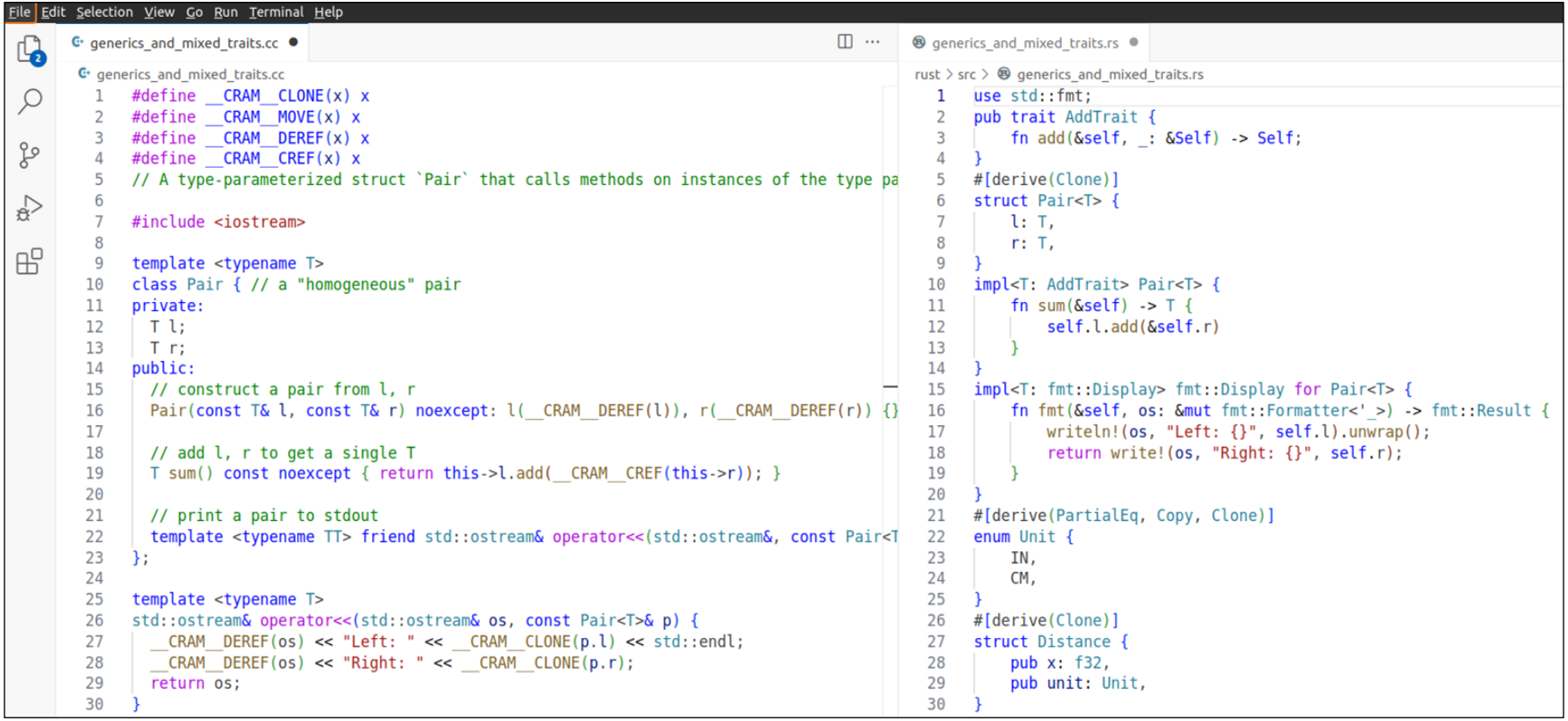
```cpp
std::list<Point> trim_front(std::list<Point>& pts, float dist) {
  if (pts.size() < 2)
    return {};

  std::list<Point> result;
  result.push_back(pts.front());
  double d = 0.0f;
  for (auto p1 = pts.begin(), p2 = std::next(pts.begin()); p2 != pts.end(); ++p1, ++p2) {
    Point& next_point = *p2;
    double segdist = p1->Distance(next_point);
    if ((d + segdist) > dist) {
      double frac = (dist - d) / segdist;
      auto midpoint = p1->PointAlongSegment(next_point, frac);
      result.push_back(midpoint);

      pts.erase(pts.begin(), p1);
      pts.front() = midpoint;
      return result;
    } else {
      d += segdist;
      result.push_back(*p2);
    }
  }

  pts.clear();
  return result;
}
```

C++

Rust

```rust
fn trim_front(pts: &mut Vec<Point>, dist: f32) -> Vec<Point> {
    if pts.len() < 2 {
        return vec![];
    };
    let mut result: Vec<Point> = Vec::new();
    result.push(pts[0].clone());
    let mut d: f64 = 0.0f32 as f64;
    for i in 0..(pts.len() - 1) {
        let next_point: &Point = &pts[i + 1];
        let segdist: f64 = pts[i].distance(&next_point);
        if (d + segdist) > (dist as f64) {
            let frac: f64 = ((dist as f64) - d) / segdist;
            let midpoint: Point = pts[i].point_along_segment(&next_point, frac);
            result.push(midpoint.clone());
            pts.drain(0..i);
            pts[0] = midpoint;
            return result;
        } else {
            d += segdist;
            result.push(pts[i + 1].clone());
        };
    }
    pts.clear();
    result
}
```

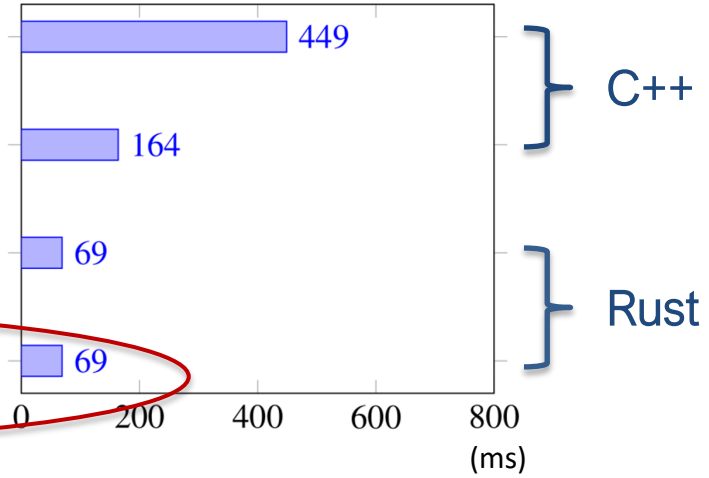
User Interface



Performance

Example: runtime comparisons for a traversal of a (large) `list<Point>`:

- original
- after hardening refactorings
- after translation to Rust by an expert
- after translation to Rust using CRAM



Potential for faster performance

Equivalent to expert manual migration

Demonstration

Discussion

Availability

- GrammaTech Services Offerings to migrate to Memory Safe
- Talk with us!
  - More detailed capability discussion
  - Solution Brief: grammatech.com/cyber-security-solutions/migration-to-memory-safe-code/
  - Email: cram@grammatech.com

Thank You!

Jason Rowley
jrowley@grammatech.com
m.704-941-5356

Ray DeMeo
rdemeo@grammatech.com
m.978-549-1122

www.grammatech.com