Safety and security

Systems programming // Trends and perspectives // ESE Kongress 2023

Kris van Rens

What’s ahead?

- The Tides They Are A-Changin’
- The current state of affairs
- What will the future bring?

A little bit about me

high tech institute

ViNotion
Making surveillance smart

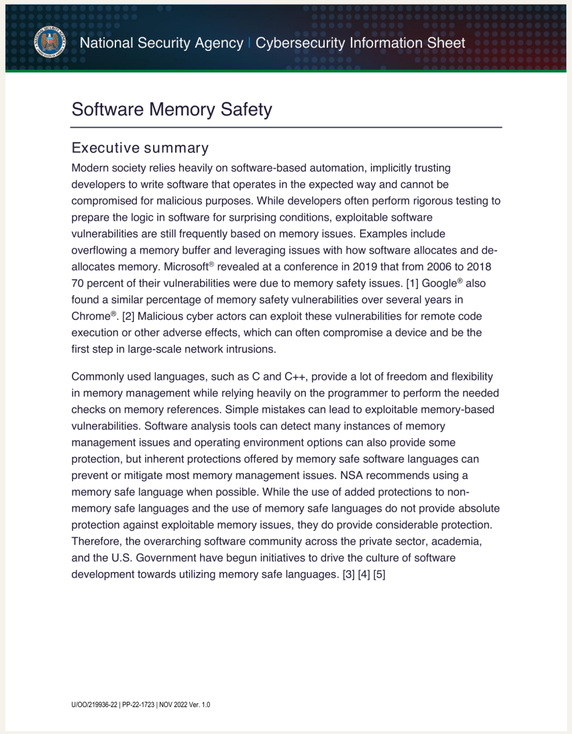
kris@vanrens.org

The Tides They Are A-Changin’


SAFETY!
CORRECTNESS!
SECURITY!
?
?
?

Stirrings (1)

Increasing law and requirement strictness



National Security Agency | Cybersecurity Information Sheet

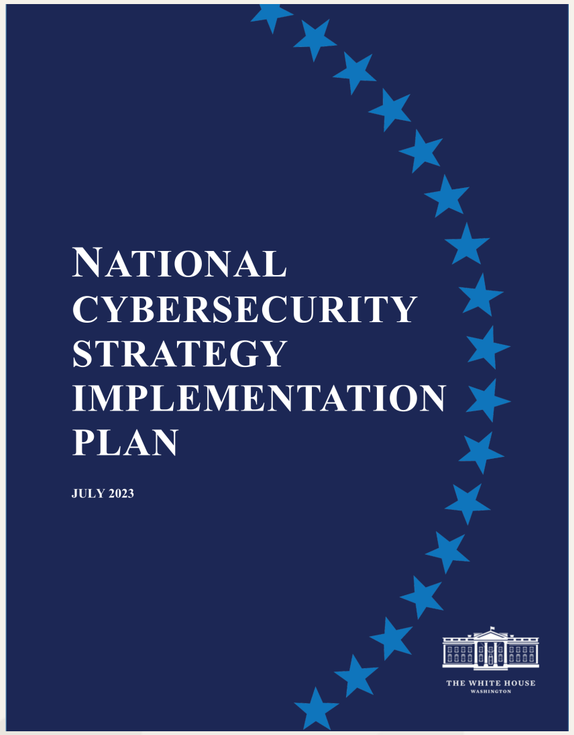
Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection. Therefore, the overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages. [3] [4] [5]

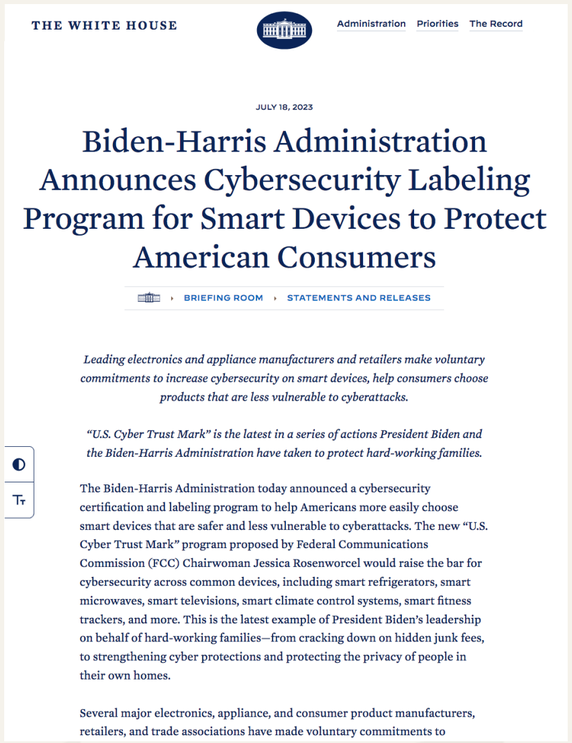
U/OO/219936-22 | PP-22-1723 | NOV 2022 Ver. 1.0



NATIONAL CYBERSECURITY STRATEGY IMPLEMENTATION PLAN

JULY 2023

THE WHITE HOUSE
WASHINGTON



THE WHITE HOUSE

Administration Priorities The Record

JULY 18, 2023

Biden-Harris Administration Announces Cybersecurity Labeling Program for Smart Devices to Protect American Consumers

BRIEFING ROOM › STATEMENTS AND RELEASES

Leading electronics and appliance manufacturers and retailers make voluntary commitments to increase cybersecurity on smart devices, help consumers choose products that are less vulnerable to cyberattacks.

"U.S. Cyber Trust Mark" is the latest in a series of actions President Biden and the Biden-Harris Administration have taken to protect hard-working families.

The Biden-Harris Administration today announced a cybersecurity certification and labeling program to help Americans more easily choose smart devices that are safer and less vulnerable to cyberattacks. The new "U.S. Cyber Trust Mark" program proposed by Federal Communications Commission (FCC) Chairwoman Jessica Rosenworcel would raise the bar for cybersecurity across common devices, including smart refrigerators, smart microwaves, smart televisions, smart climate control systems, smart fitness trackers, and more. This is the latest example of President Biden's leadership on behalf of hard-working families—from cracking down on hidden junk fees, to strengthening cyber protections and protecting the privacy of people in their own homes.

Several major electronics, appliance, and consumer product manufacturers, retailers, and trade associations have made voluntary commitments to

enisa

EUROPEAN UNION AGENCY FOR CYBERSECURITY

ADVANCING SOFTWARE SECURITY IN THE EU

The role of the EU cybersecurity certification framework

NOVEMBER 2019

Stirrings (2)

Existing/upcoming languages have a solid story

Stirrings (3)

Tool / language development rate skews

- It often takes > a decade to get a substantial feature into C or C++,
- Backwards compatibility can be considered to impede innovation,
- Newer languages have a community-driven, faster release model,
- Western-originated administrative bodies like ISO are non-inclusive.

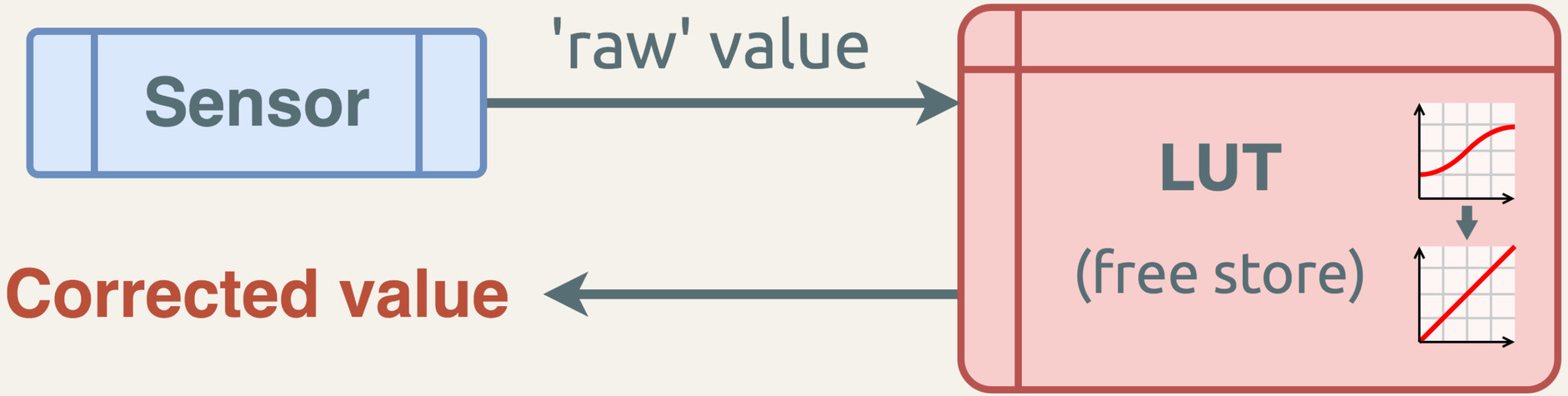
Language safety

- **Arithmetic safety** (e.g. integer overflow, floating point value handling)
- **Thread safety** (e.g. data sharing in concurrent contexts)
- **Definition safety** (e.g. ODR violation)
- **Memory safety** ⇐ **main focus of discussions on language safety**
  - *Initialization safety* (e.g. zero initialization or forced initialization)
  - *Type safety* (e.g. strong typing, type punning)
  - *Bounds safety* (e.g. array index checking, pointer arithmetic)
  - *Lifetime safety* (e.g. ownership model and lifetime verification)

The current state of affairs


Audits
Static analysis
Study
Training
Compiler flags
LINT
AI
Regulations
Exercises
Reviews
Best practices
Guidelines
Tests
Debugging
Design patterns
Documentation
Idioms
Compiler sanitizers

Examples introduction: LUT



Read while value below `THRESHOLD`, for a maximum number of `MAX_ITER` times

Examples (1): Creating the LUT

```c
void load_values(float* const lut, size_t s);

float* create_lut() {
  const size_t SIZE = 8192;

  float* lut = malloc(sizeof(float) * SIZE);
  if (lut) {
    load_values(lut, SIZE);
  }

  return lut;
}

float* const lut = create_lut();
if (!lut) {
  // ..handle error..
}

// ..do stuff with LUT..

free(lut);
```

```cpp
void load_values(std::span<float> lut) noexcept;

[[nodiscard]] std::vector<float> create_lut() {
  auto lut = std::vector<float>(8192);

  load_values(lut);

  return lut;
}

try {
  const auto lut = create_lut();

  // ..do stuff with LUT..

} catch(...) {
  // ..handle error..
}
```

Examples (1): Creating the LUT

```cpp
void load_values(std::span<float> lut) noexcept;

[[nodiscard]] std::vector<float> create_lut() {
  auto lut = std::vector<float>(8192);

  load_values(lut);

  return lut;
}

try {
  const auto lut = create_lut();

  // ..do stuff with LUT..

} catch(...) {
  // ..handle error..
}
```

Errors propagate as exceptions

```rust
fn load_values(lut: &mut [f32]);

fn create_lut() -> Result<Vec<f32>> {
  const SIZE: usize = 8192;

  let mut lut = Vec::new();
  lut.try_reserve_exact(SIZE)?;
  lut.resize(SIZE, 0f32);

  load_values(&mut lut);

  Ok(lut)
}

let Ok(lut) = create_lut() else {
  // ..handle error..
};

// ..do stuff with LUT..
```

Errors propagate as `Result` values

Examples (2): Using the LUT

```c
float* create_lut();
int read_sensor(); // Returns <0 when an error occurs.
size_t value2index(int value);

enum Error { ERR_OK = 0, ERR_SENSOR_READ } error = ERR_OK;

float* const lut = create_lut();
if (!lut) {
  // ..handle error..
}

for (int i = 0; i < MAX_ITERS; i++) {
  const int value = read_sensor();
  if (value < 0) {
    error = ERR_SENSOR_READ;
    break;
  }

  if (lut[value2index(value)] > THRESHOLD) {
    break;
  }

  // ..sleep for a bit..
}

switch (error) { /* ..handle error + free LUT.. */ }
```

```cpp
[[nodiscard]] std::vector<float> create_lut();
[[nodiscard]] std::optional<int> read_sensor() noexcept;
[[nodiscard]] constexpr std::size_t value2index(int value) noexcept;

try {
  const auto lut = create_lut();

  const auto is_sensor_low = [&](int) -> bool {
    const auto value = read_sensor();
    if (!value) {
      throw std::system_error{EIO, std::generic_category()};
    }

    return lut.at(value2index(*value)) < THRESHOLD;
  };

  using namespace std::views;
  for (int _ : iota(0, MAX_ITERS) | take_while(is_sensor_low)) {
    // ..sleep for a bit..
  }
} catch(...) {
  // ..handle error..
}
```

Examples (2): Using the LUT

```cpp
[[nodiscard]] std::vector<float> create_lut();
[[nodiscard]] std::optional<int> read_sensor() noexcept;
[[nodiscard]] constexpr std::size_t value2index(int value) noexcept;

const auto lut = create_lut();

const auto is_sensor_low = [&](int) -> bool {
  const auto value = read_sensor();
  if (!value) {
    throw std::system_error{EIO, std::generic_category()};
  }

  return lut.at(value2index(*value)) < THRESHOLD;
};

using namespace std::views;
for (int _ : iota(0, MAX_ITERS) | take_while(is_sensor_low)) {
  // ..sleep for a bit..
}
```

```rust
fn create_lut() -> Result<Vec<f32>>;
fn read_sensor() -> Result<i32>;
const fn value2index(value: i32) -> usize;

let lut = create_lut()?;

let is_sensor_low = |_| -> Result<bool> {
  Ok(lut[value2index(read_sensor()?)] < THRESHOLD)
};

let sleep = || { /* ..sleep for a bit.. */ };

(0..MAX_ITERS).try_take_while(is_sensor_low, sleep)?;
```

(`try_take_while` is a custom iterator adapter)

Digression: Undefined Behavior

Wait...programming is not exact science?

Indexing out-of-bounds:

```c
int array[8] = {};
int x = array[10]; // ???
```

Signed integer overflow:

```c++
int x = std::numeric_limits<int>::max();
int y = x + 5; // ???
```

Accessing uninitialized values:

```c
int x;
printf("%d\n", x); // ???
```

Shared mutable access of data:

```c++
std::string x;
std::jthread t1{[&]{ x = "Uh"; }};
std::jthread t2{[&]{ x = "Oh"; }};
std::cout << x << '\n'; // ???
```

UB provides “wiggle room” for compiler writers, but can be dangerous

Digression: Undefined Behavior

Safe languages don’t have UB

Indexing out-of-bounds:

```rust
let x = [0; 8];
let y = x[10]; // Won't compile, or panics during runtime.
```

Signed integer overflow:

```rust
let x = i32::MAX;
let y = x + 1; // Won't compile, or panics during runtime.
```

Accessing uninitialized values:

```rust
let x;
println!("{x}"); // Won't compile.
```

Shared mutable access of data:

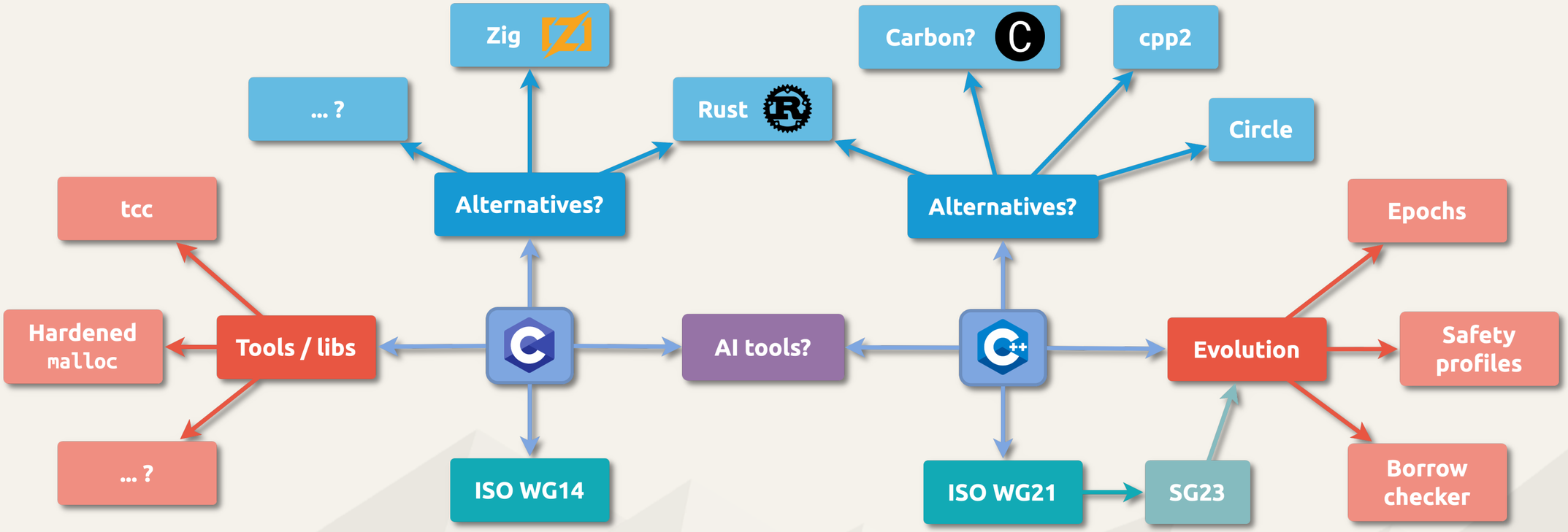
```rust
let mut x = String::new();
let t1 = thread::spawn(||{ x = "Uh".to_owned(); });
let t2 = thread::spawn(||{ x = "Oh".to_owned(); });
println!("{x}"); // Won't compile.
```

Safe languages make you pay correctness costs up front, preventing issues later

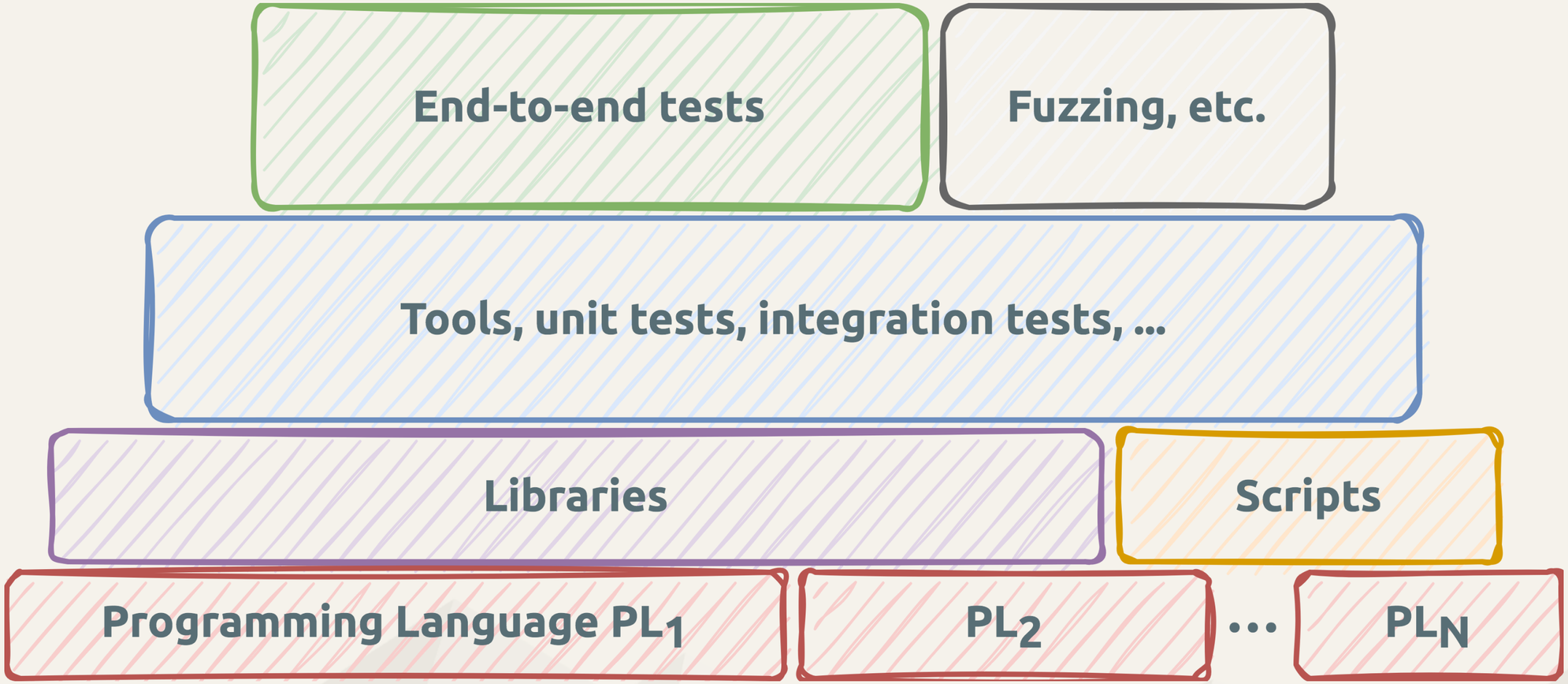
What will the future bring?

The point is made ☝

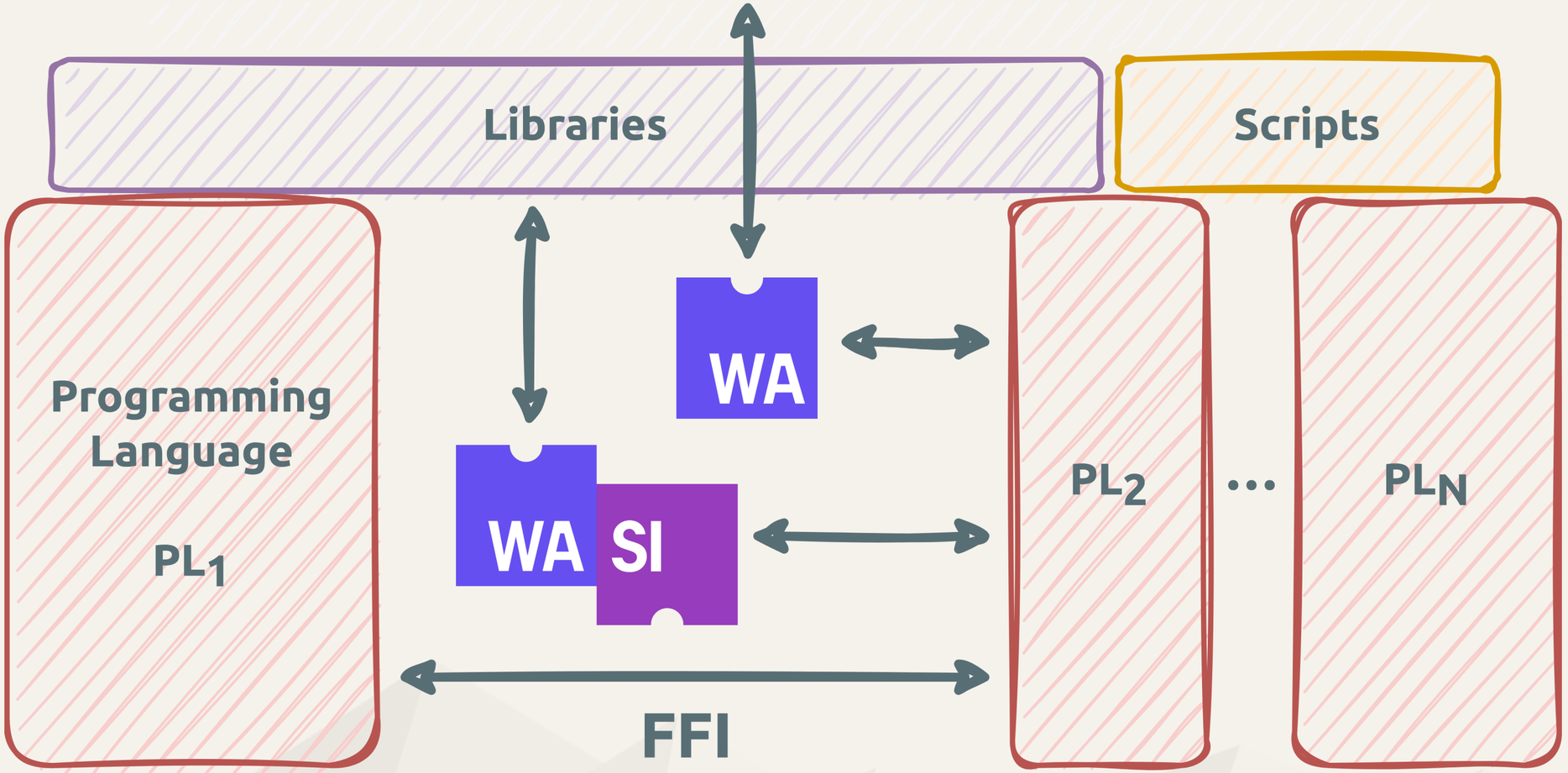
Language safety is a hot and contentious topic



What about tools? 🛠️



..and what about legacy code?



Benefit from safe languages today

- Try safe tools like Rust,
- Stay ahead of the curve,
- Level-up programming skills,
- Adopt step-by-step.

High Tech Institute:

Visit the High Tech Institute website for more information.

Concluding remarks

High Tech Institute trainings

C++ fundamentals 

A comprehensive intro course to C++20

For developers of all levels, with prior programming experience (any lang.).

Four full days in total – 2x two days within two weeks – ±50% exercises.

Goal: Provide a solid base of C++20 knowledge, effective handles for further development and a successful career as a professional.

Course information @ High Tech Institute.

Exploring Rust 

A thorough introduction course to Rust

For developers with solid prior programming experience (any lang.).

Nine hours total – two half-days within two weeks – ±30% exercises.

Goal: Provide an experienced software engineer the necessary knowledge to build a strong value judgment of the Rust language.

Course information @ High Tech Institute.

End

Thank you 😀

high
tech
institute

github.com/krisvanrens