Memory Safety for large C/C++ codebases

Strategies and techniques

Florian Noeding

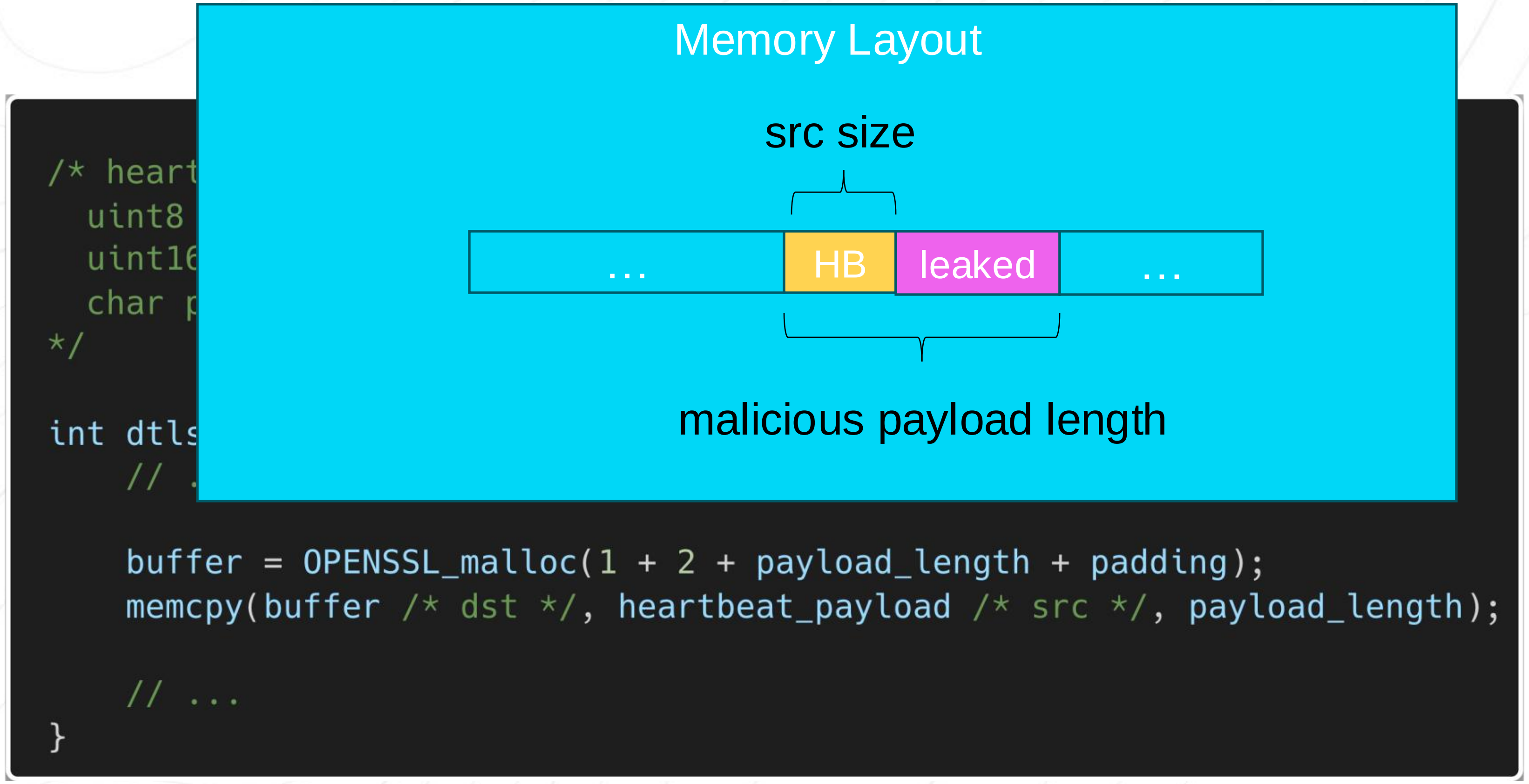
Principal, Security Architect

Adobe

https://florian.noeding.com/

Memory Safety Vulnerabilities

Heartbleed
CVE-2014-0160
Out-of-bounds read in
OpenSSL



```
/* heart
  uint8
  uint16
  char p
*/

int dtls
    // .

    buffer = OPENSSL_malloc(1 + 2 + payload_length + padding);
    memcpy(buffer /* dst */, heartbeat_payload /* src */, payload_length);

    // ...
}
```

Why memory safety?

Largest class of CVEs affecting C/C++ based software

Experts struggle to write safe code in C/C++

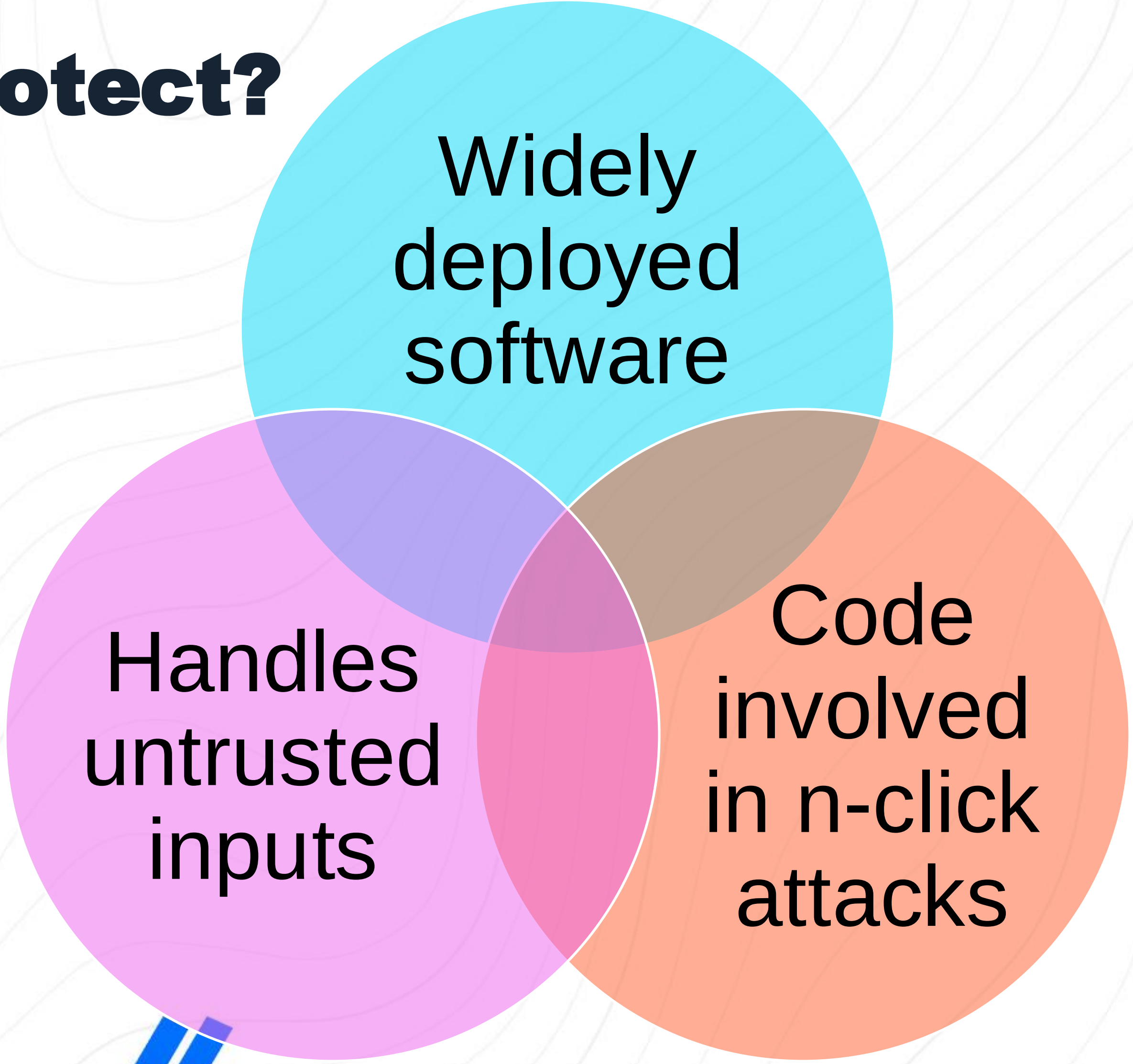
C and C++ are ubiquitous

Looming regulatory pressure

United States National Cyber Security Strategy 2023 , page 19

To address these challenges, the Administration will shape the long-term security and resilience of the digital ecosystem, against both today's threats and tomorrow's challenges. We must hold the stewards of our data accountable for the protection of personal data; drive the development of more secure connected devices; and reshape laws that govern liability for data losses and harm caused by cybersecurity errors, software vulnerabilities, and other risks created by software and digital technologies. We will use Federal purchasing power and grant-making to incentivize security. And we will explore how the government can stabilize insurance markets against catastrophic risk to drive better cybersecurity practices and to provide market certainty when catastrophic events do occur.

What to protect?

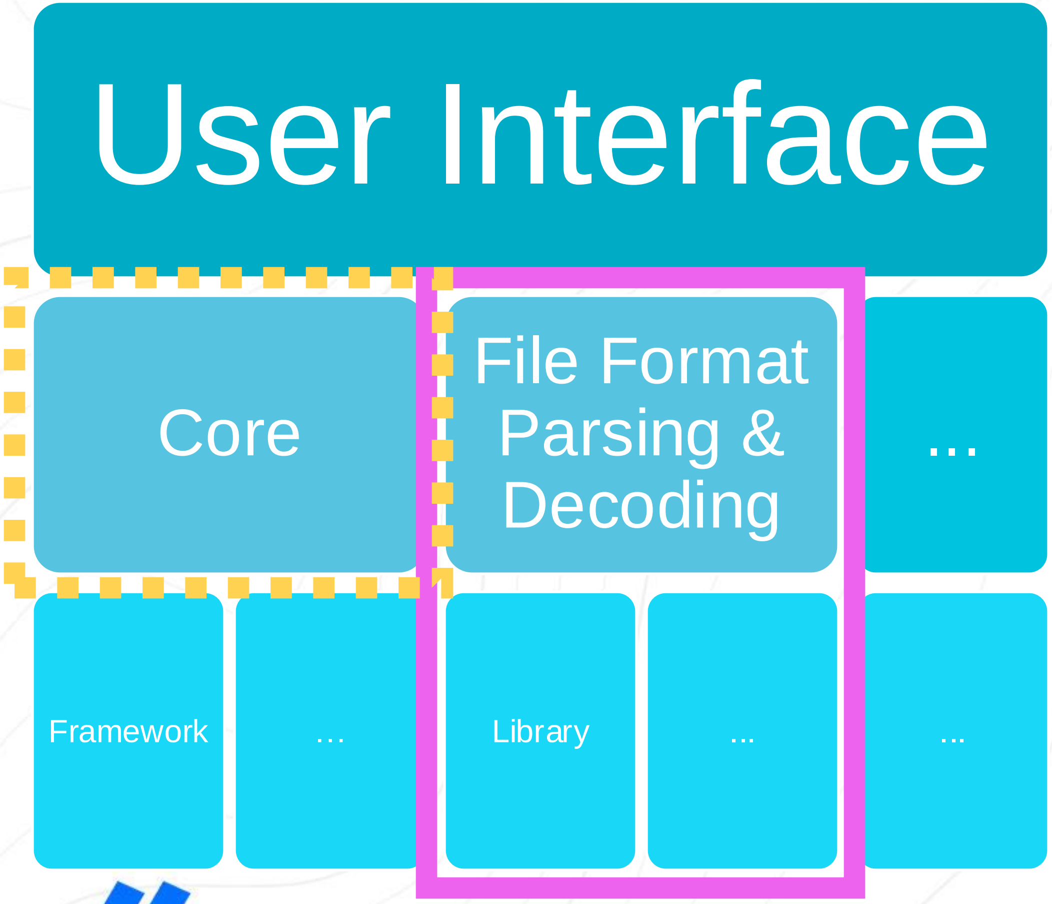


Widely deployed software

Handles untrusted inputs

Code involved in n-click attacks

Prioritizing Risks in an Abstract Desktop App



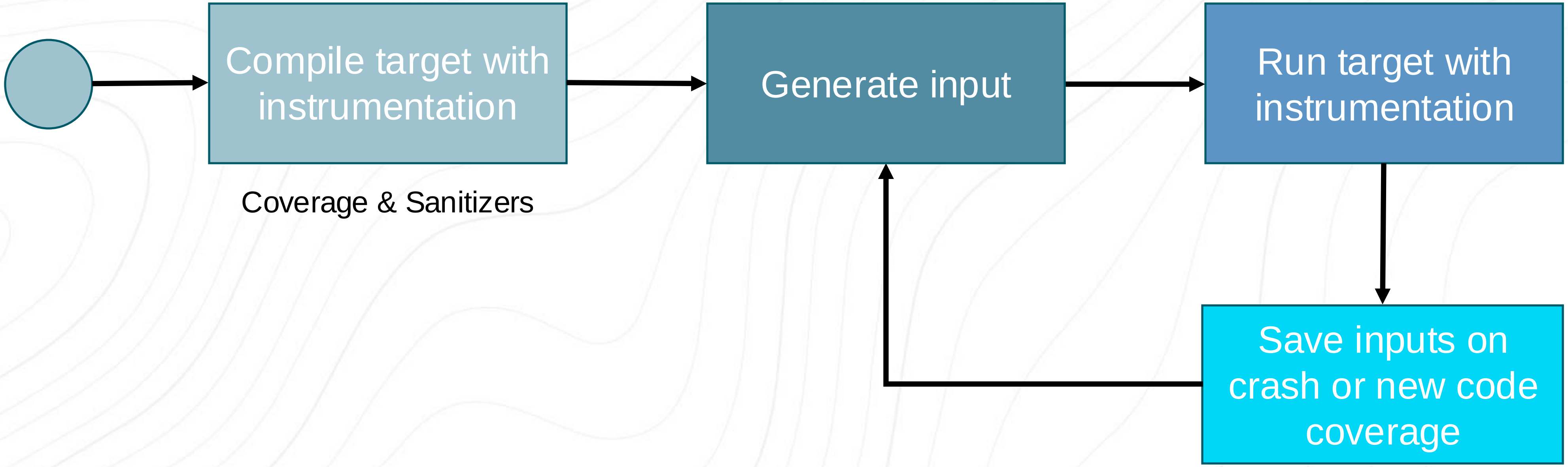
In-memory representation of data

(and dynamic code if relevant)

Document, Image, Video, Audio, Network, …

Strategy 1: Fuzzing

Fuzzing: coverage guided testing with semi-random data



Compile target with instrumentation

Coverage & Sanitizers

Generate input

Run target with instrumentation

Save inputs on crash or new code coverage

Fuzzing Heartbleed: idea

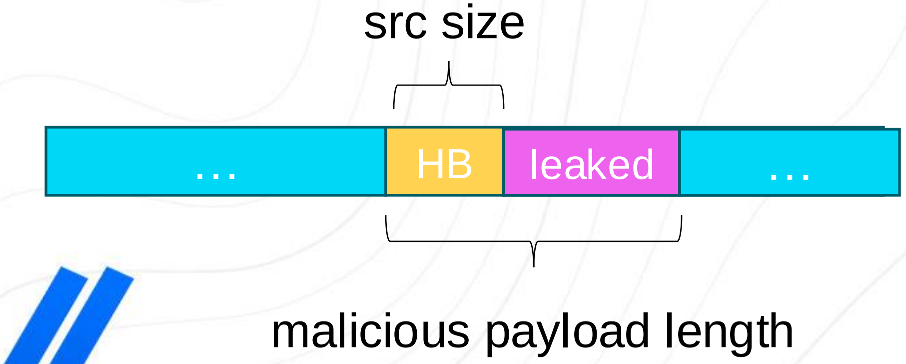
```
/* heartbeat message
  uint8 type
  uint16 payload_length
  char payload[$SIZE]
*/

int dtls1_process_heartbeat(SSL *s) {
    // ...

    buffer = OPENSSL_malloc(1 + 2 + payload_length + padding);
    memcpy(buffer /* dst */, heartbeat_payload /* src */, payload_length);

    // ...
}
```

Input Data

0x18 0x00FF

→ out-of-bounds read

type payload_length

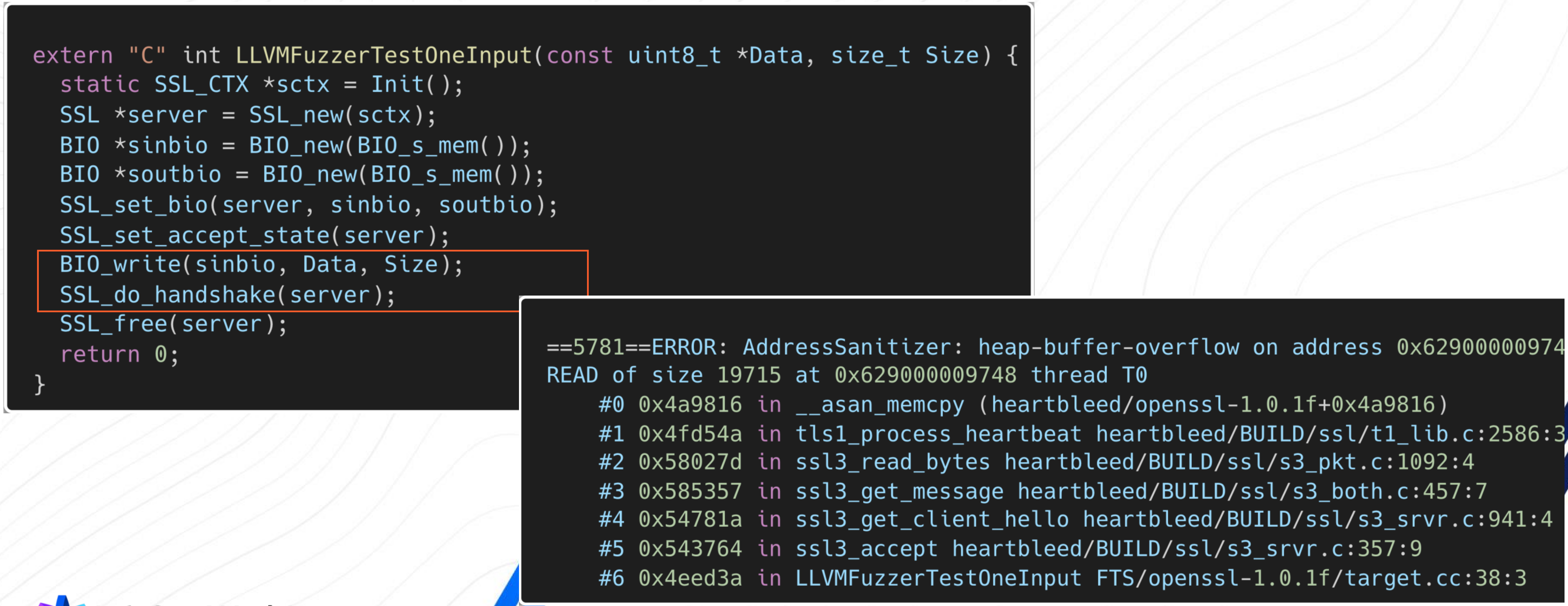


src size

…	HB	leaked	…

malicious payload length

Fuzzing Heartbleed: code

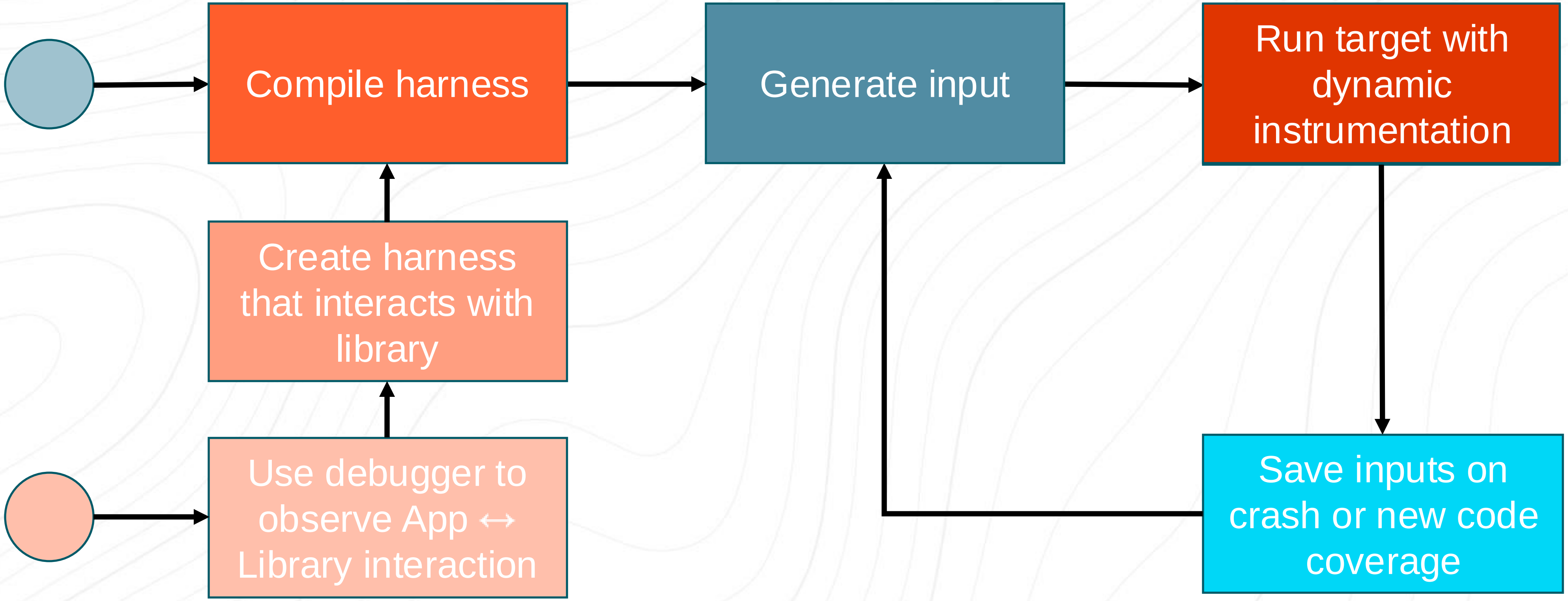


```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  static SSL_CTX *sctx = Init();
  SSL *server = SSL_new(sctx);
  BIO *sinbio = BIO_new(BIO_s_mem());
  BIO *soutbio = BIO_new(BIO_s_mem());
  SSL_set_bio(server, sinbio, soutbio);
  SSL_set_accept_state(server);
  BIO_write(sinbio, Data, Size);
  SSL_do_handshake(server);
  SSL_free(server);
  return 0;
}
```

```
==5781==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x62900000974
READ of size 19715 at 0x629000009748 thread T0
    #0 0x4a9816 in __asan_memcpy (heartbleed/openssl-1.0.1f+0x4a9816)
    #1 0x4fd54a in tls1_process_heartbeat heartbleed/BUILD/ssl/t1_lib.c:2586:3
    #2 0x58027d in ssl3_read_bytes heartbleed/BUILD/ssl/s3_pkt.c:1092:4
    #3 0x585357 in ssl3_get_message heartbleed/BUILD/ssl/s3_both.c:457:7
    #4 0x54781a in ssl3_get_client_hello heartbleed/BUILD/ssl/s3_srvr.c:941:4
    #5 0x543764 in ssl3_accept heartbleed/BUILD/ssl/s3_srvr.c:357:9
    #6 0x4eed3a in LLVMFuzzerTestOneInput FTS/openssl-1.0.1f/target.cc:38:3
```

How adversaries find memory safety flaws

Adversaries can use binary fuzzing



Compile harness

Generate input

Run target with dynamic instrumentation

Create harness that interacts with library

Use debugger to observe App ↔ Library interaction

Save inputs on crash or new code coverage

Why fuzzing?

Pros

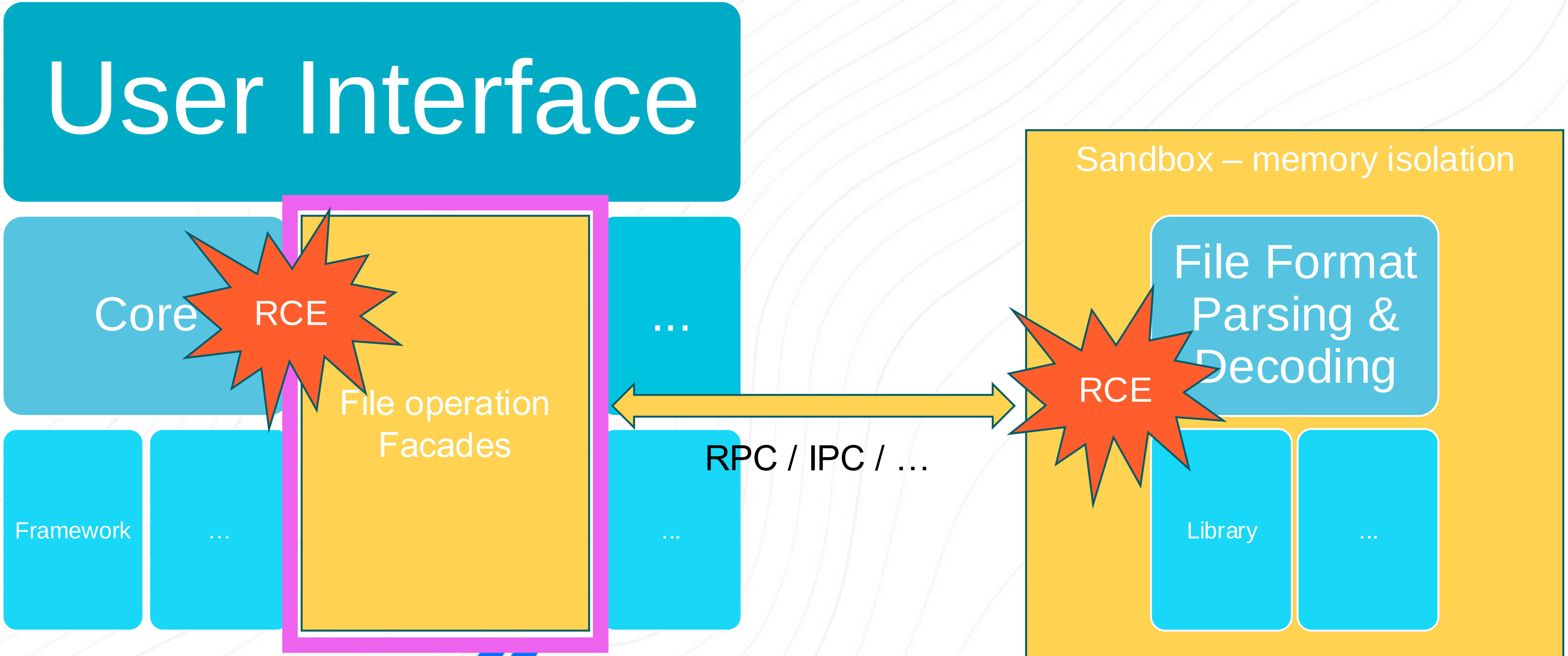
- 🟢 No changes to shipped product needed
- 🧪 For library code: Feels like writing unit tests → developer friendly
- 📊 Helps estimate memory safety challenge size
- 👹 Adversaries use the same technique

Cons

- 🧗♂ Fuzzing applications is much harder than libraries
- 🚨 Reactive, can’t guarantee absence of memory safety flaws
- 🤯 Can feel like whack-a-mole to engineers

Strategy 2: Sandboxing

Fault isolation as defense in depth



Two common approaches

Process Isolation

- review Chromium’s model
- Minimize privileges per process using operating system controls

Memory Isolation via transpilation

- review Firefox’s model; RLBox
- Run library code in WASM interpreter, providing privilege and memory isolation

Why sandboxing?

Pros

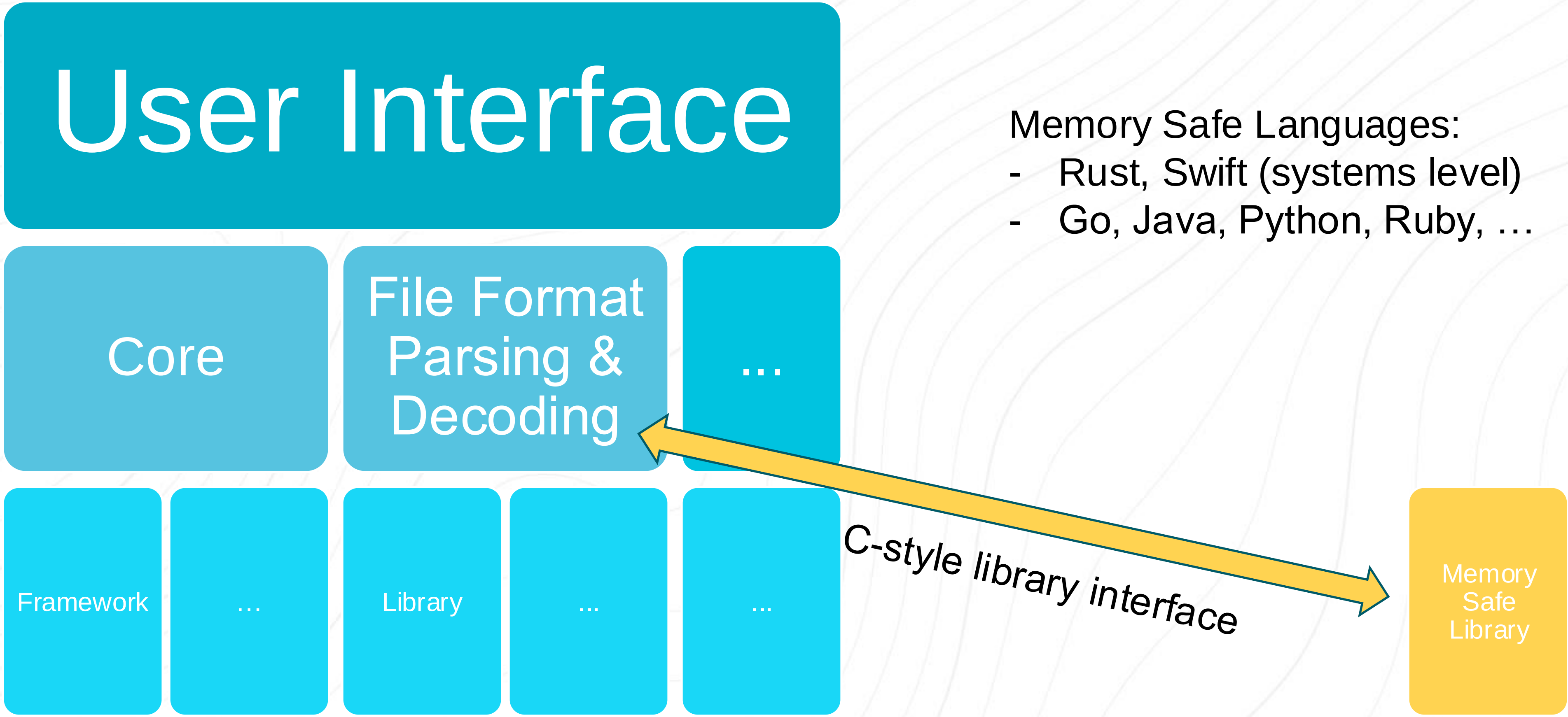
- 📦 Mitigate memory safety flaws of **existing** C/C++ code
- 🛡️ Pro-active mechanism with safety guarantees
- 🎯 Enables focus on sandbox and interface layers for security reviews

Cons

- 💤 Performance overhead
- 🚫 Platform compatibility issues
- 🤯 Challenges with debugging across sandbox layer, hard to retrofit (process based sandboxing)
- 🧩 Interaction heavy code is hard to sandbox
- 🧨 Introduces new risks at the boundary layer

Strategy 3: Rewriting

Eliminating memory safety flaws by design



Memory Safe Languages:

- Rust, Swift (systems level)
- Go, Java, Python, Ruby, …

Know how to solve Rust ↔ C++?
Talk to me, please!

Why rewrite?

Pros

- 🛡️ Mitigates memory safety flaws **by design**
- 🚀 Developer productivity benefits: modern ecosystem with packages and additional correctness guarantees
- 🎯 Enables focus on interface layers and unsafe code for security reviews

Cons

- 🚫 Depending on language, platform compatibility issues (Swift, not Rust)
- 🤯 Bi-directional interaction between language boundaries is challenging, especially when combining C++ and Rust
- 🧨 Introduces new risks at the boundary layer
- 🎓 Engineers must learn new paradigms (e.g., Rust’s borrow checker)

Further Strategies

Tech is relatively easy
-
Driving change is hard

Step 1: Is memory safety truly a top business concern?

- Limited engineering bandwidth
- Focus
- Provide relevant data

Step 2: Seek Partners in Engineering

- Partner with Staff+ / Principal Engineers
- Earn trust & buy-in for memory safety
- Security: prioritize
- Engineering: technical approach

Step 3: Frame the conversation in business terms

- Looming threat of regulation?
- What is your competition doing?
- Productivity benefits of adopting memory safe languages
- Risks to customers and the business

Step 4: Get it on the roadmap

- Build consensus with leadership
- Dual strategy: bottom-up and top-down
- Evangelize, Evangelize, Evangelize

Step 5: Execute iteratively

- Always keep shipping
- Iterative instead of big-bang rewrites

This is a toolkit to create YOUR strategy

A mental model to prioritize security risks

Strategies to mitigate the highest risks

One way to drive organizational change

https://florian.noeding.com/

THANK YOU!

References in appendix of PDF

Appendix

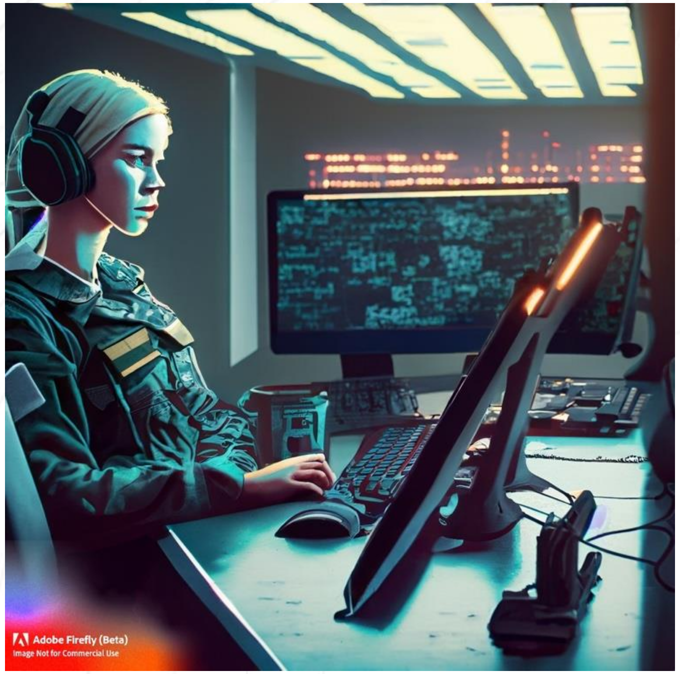
Adversary Model (simplified)

Researchers

- Generally friendly
- Various motivations

eCriminals

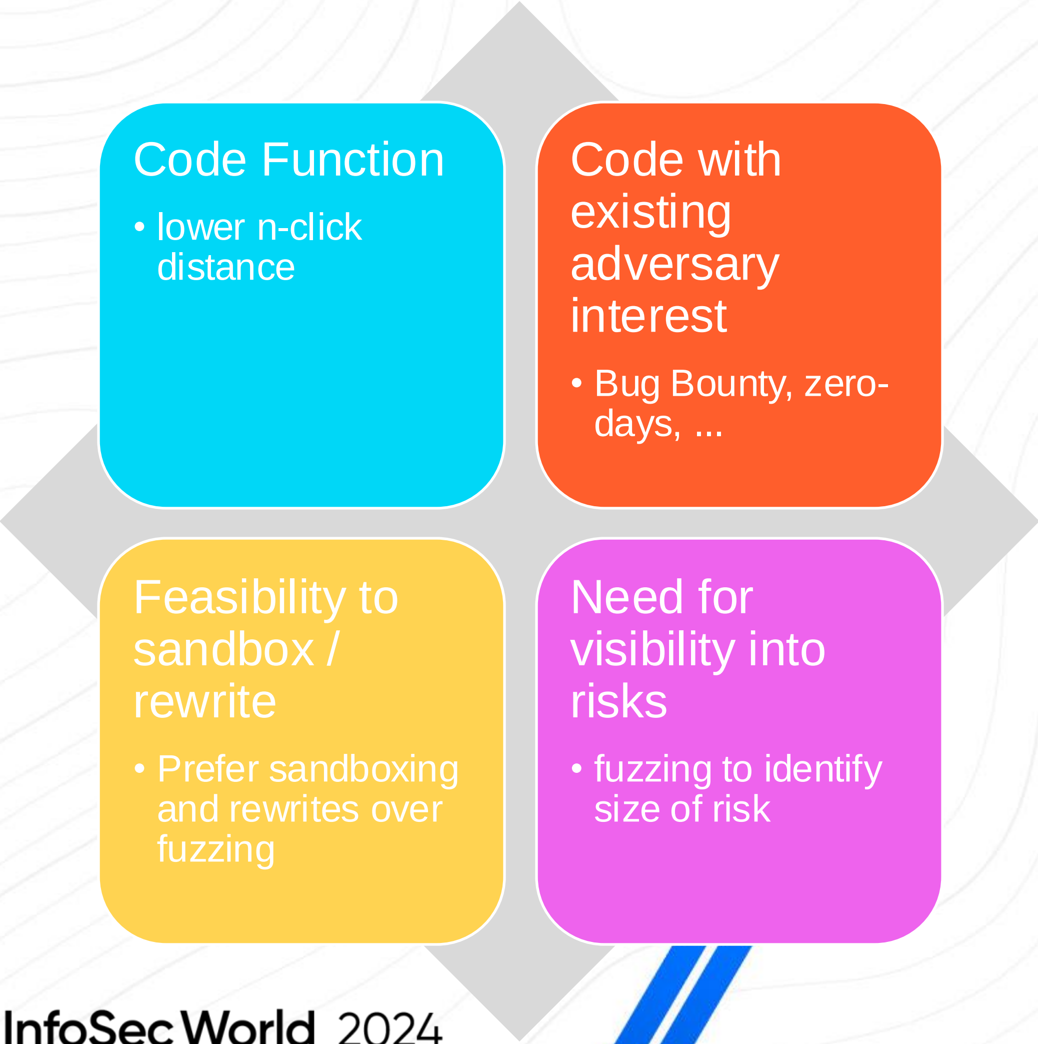
- Financially Motivated
- Strategy: repeatable and scalable exploitation



Nation States

- Targeted operations
- Hard to predict

Prioritization approaches within YOUR threat model



Code Function

- lower n-click distance

Code with existing adversary interest

- Bug Bounty, zero-days, ...

Feasibility to sandbox / rewrite

- Prefer sandboxing and rewrites over fuzzing

Need for visibility into risks

- fuzzing to identify size of risk

Example strategy

1. Focus on file parsing and decoding libraries
2. Integrate a generic sandbox to mitigate risks during file parsing and decoding
3. Write new file parsing libraries in Rust with a C-style interface
4. Prioritize code for fuzzing, where sandboxing and rewriting is not feasible.

References: Why memory safety?

- 70% of CVEs assigned by Microsoft are due to memory safety flaws
  - https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/
- 70% of CVEs in Chromium are due to memory safety flaws
  - https://www.chromium.org/Home/chromium-security/memory-safety/
- 94% of critical and high CVEs in Mozilla are due to memory safety flaws
  - https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/
- United States Cyber Security Strategy
  - https://www.whitehouse.gov/wp-content/uploads/2023/03/National-Cybersecurity-Strategy-2023.pdf
- CISA: The Urgent Need for Memory Safety in Software Products
  - https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products
- CISA: The Case for Memory Safety Roadmaps
  - https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf
- Adobe’s response to RFI CISA-2023-0027 (search for memory safety)
  - https://downloads.regulations.gov/CISA-2023-0027-0063/attachment_1.pdf

References: What to protect?

- Chromium rule of 2
  - https://chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md
- The SUX Rule for Safer Code:
  - https://kellyshortridge.com/blog/posts/the-sux-rule-for-safer-code/

References: Fuzzing

- Google’s libFuzzer Tutorial (includes the Heartbleed example)
  - https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md
- Fuzzers:
  - libFuzzer: https://llvm.org/docs/LibFuzzer.html
  - AFLplusplus: https://github.com/AFLplusplus/AFLplusplus
  - Hongfuzz: https://github.com/google/honggfuzz
- Binary fuzzing:
  - https://medium.com/@kciredor/fuzzing-adobe-reader-for-exploitable-vulns-fun-profit-76edb6a5b012
  - https://research.checkpoint.com/2018/50-adobe-cves-in-50-days/
  - https://gosecure.ai/blog/2019/07/30/fuzzing-closed-source-pdf-viewers/
  - https://bushido-sec.com/index.php/2023/06/25/the-art-of-fuzzing-windows-binaries/
- Trail of Bits Testing Handbook - Fuzzing guide
  - https://appsec.guide/docs/fuzzing/

References: Sandboxing

- RLBox (using WASM)
  - Website: https://rlbox.dev/ (review references at bottom of page)
  - https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/
  - https://blog.mozilla.org/attack-and-defense/2021/12/06/webassembly-and-back-again-fine-grained-sandboxing-in-firefox-95/
- Sandboxing on Linux using seccomp (limit process privileges)
  - https://blog.cloudflare.com/sandboxing-in-linux-with-zero-lines-of-code/

References: Rewriting

- Systems programming languages
  - Rust: https://www.rust-lang.org/
  - Swift: https://developer.apple.com/swift/
  - Hylo (formerly Val): https://www.hylo-lang.org/ (experimental language project supported by Adobe)
- Secure By Desing – Google's Perspective on Memory Safety https://security.googleblog.com/2024/03/secure-by-design-googles-perspective-on.html