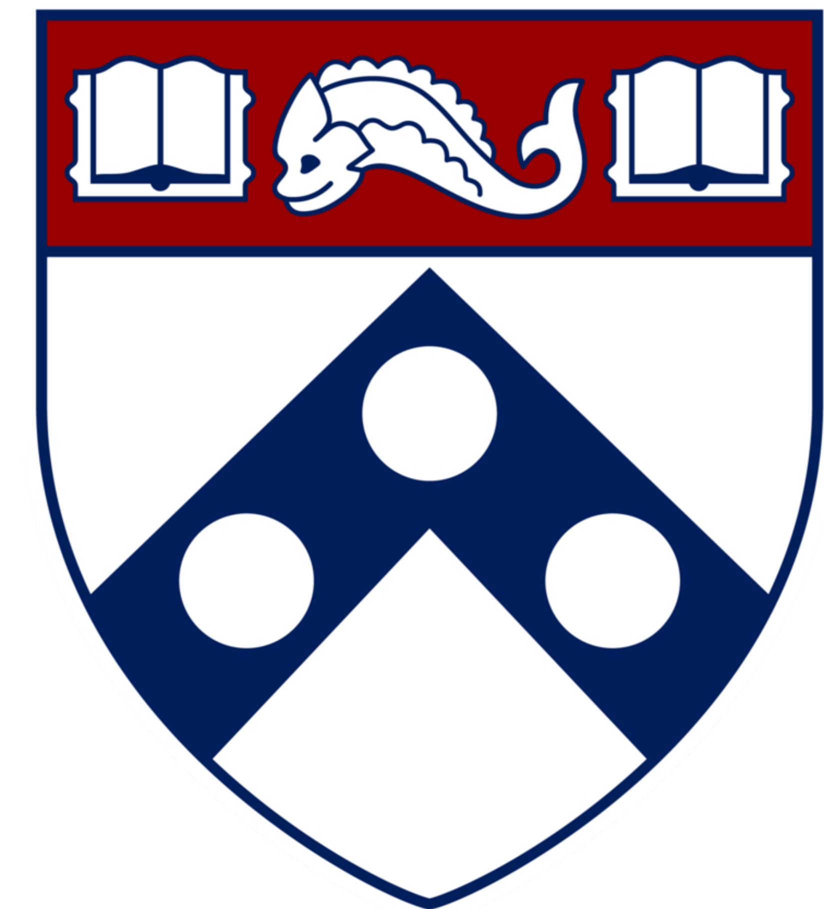
Secure Systems Engineering and Management

A Data-driven Approach



Memory Safe Programming

Michael Hicks

UPenn CIS 7000-003
Spring 2026

What is memory safety? - The

www.pl-enthusiast.net/2014/07/21/memory-safety/

The Programming Languages Enthusiast

HOME ABOUT THE PL ENTHUSIAST

← Program verification in the undergraduate CS curriculum

Spotlight: Ravi Chugh →

BY MICHAEL HICKS | JULY 21, 2014 · 7:09 AM

↓ Jump to Comments

What is memory safety?

I am in the process of putting together a MOOC on software security, which goes live in October. At the moment I'm finishing up material on buffer overflows, format string attacks, and other sorts of vulnerabilities in C. After presenting this material, I plan to step back and say, "What do these errors have in common? They are violations of *memory safety*." Then I'll state the definition of memory safety, say why these vulnerabilities are violations of memory safety, and conversely say why memory safety, e.g., as ensured by languages like Java, prevents them.

Search

Recent Posts

- How to Write a Grad School Personal Statement
- BullFrog: Online Schema Migration, On Demand
- Increasing the Impact of PL Research
- "What is PL Research?" The Talk
- How to Write a Conference Talk

Subscribe to Blog via Email

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

http://www.pl-enthusiast.net/2014/07/21/memory-safety/

Low-level attacks enabled by a lack of **Memory Safety**

A memory safe program execution:

1. only **creates pointers** through **standard means**
   - `p = malloc(…)`, or `p = &x`, or `p = &buf[5]`, etc.
2. only uses a pointer to **access memory** that **“belongs” to that pointer**

Combines two ideas:

temporal safety and **spatial safety**

Spatial safety

- View pointers as triples (**p**,***b***,***e***)
  - **p** is the actual pointer
  - ***b*** is the base of the memory region it may access
  - ***e*** is the extent (bounds) of that region
- **Access allowed *iff* *b* ≤ p ≤ e**-`sizeof`(`typeof`(**p**))
- Operations:
  - Pointer arithmetic increments **p**, leaves ***b*** and ***e*** alone
  - Using `&`: ***e*** determined by size of original type

Examples

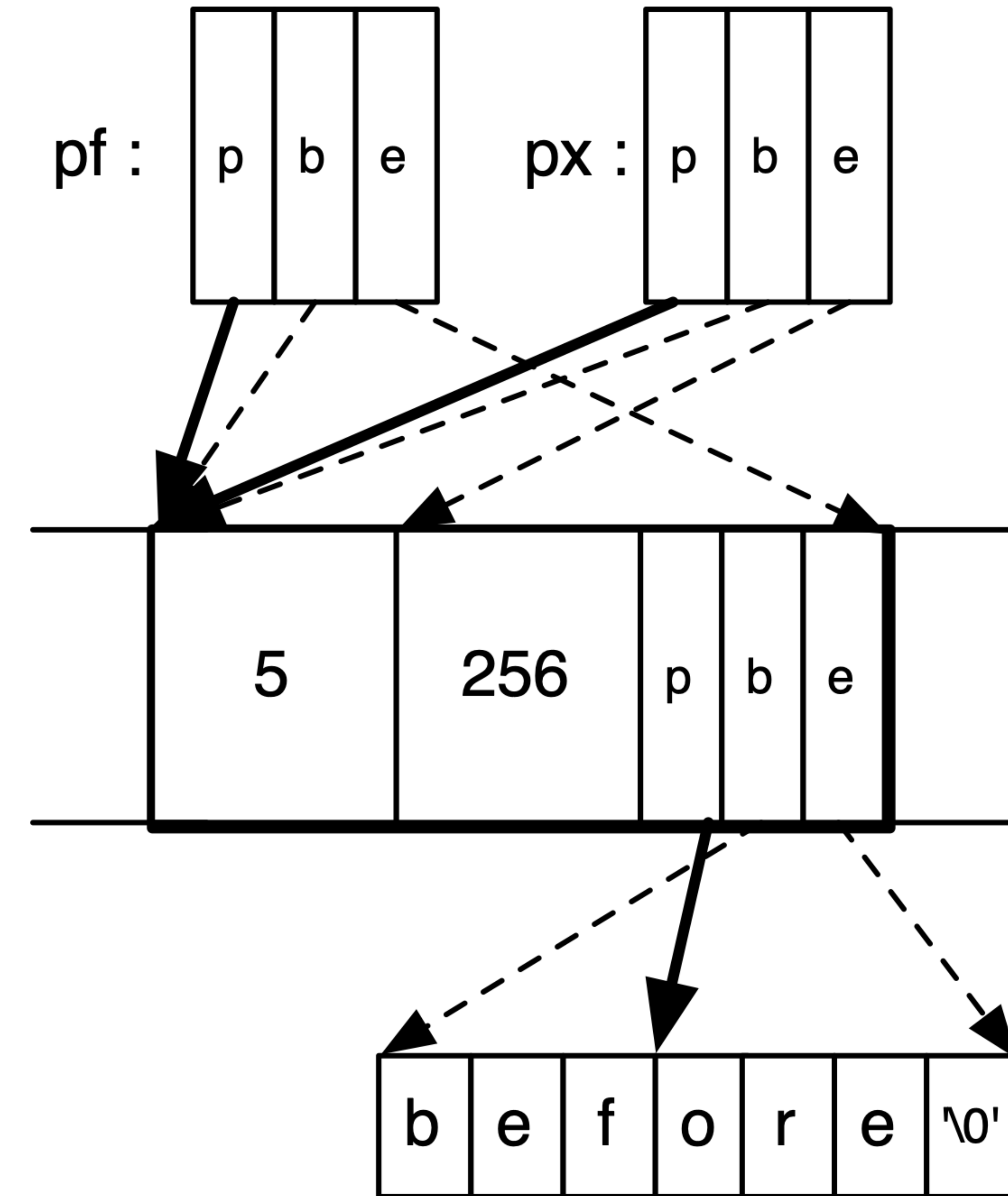
```
int x;            // assume sizeof(int)=4
int *y = &x;      // p = &x, b = &x, e = &x+4
int *z = y+1;     // p = &x+4, b = &x, e = &x+4
*y = 3;           // OK: &x ≤ &x ≤ (&x+4)-4
*z = 3;           // Bad: &x ≤ &x+4 ≰ (&x+4)-4
```

```
struct foo {
 char buf[4];
 int x;
};
```

```
struct foo f = { "cat", 5 };
char *y = &f.buf; // p = b = &f.buf, e = &f.buf+4
y[3] = 's';       // OK: p = &f.buf+3 ≤ (&f.buf+4)-1
y[4] = 'y';       // Bad: p = &f.buf+4 ≰ (&f.buf+4)-1
```

Visualized example

```
struct foo {
  int x;
  int y;
  char *pc;
};
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
int *px = &pf->x;
```



No buffer overflows

- A buffer overflow violates spatial safety

```
void copy(char *src, char *dst, int len)
{
  int i;
  for (i=0;i<len;i++) {
    *dst = *src;
    src++;
    dst++;
  }
}
```

  - Overrunning the bounds of the source and/or destination buffers implies either `src` or `dst` is illegal

No format string attacks

- The call to `printf` dereferences illegal pointers

```
char *buf = “%d %d %d\n”;
printf(buf);
```

  - View the stack as a buffer defined by the number and types of the arguments it provides
  - The extra format specifiers construct pointers beyond the end of this buffer and dereference them

- Essentially a kind of buffer overflow

Temporal safety

- A **temporal safety violation** occurs when trying to **access undefined memory**
  - Spatial safety assures it was to a legal region
  - Temporal safety assures that region is still in play
- Memory regions either **defined** or **undefined**
  - Defined means allocated (and active)
  - Undefined means unallocated, uninitialized, or deallocated
- Pretend memory is infinitely large (we never reuse it)

No dangling pointers

- Accessing a freed pointer violates temporal safety

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation
```

The memory dereferenced no longer belongs to p.

- Accessing uninitialized pointers is similarly not OK:

```
int *p;
*p = 5; // violation
```

Goes for NULL pointers too

Most languages are memory safe

- The easiest way to avoid all of these vulnerabilities is to **use a memory safe language**
- Modern languages are memory safe
  - Java, Python, C#, Ruby
  - Haskell, Scala, Go, Objective Caml, Rust
- In fact, these **languages are type safe,** which is even **stronger**

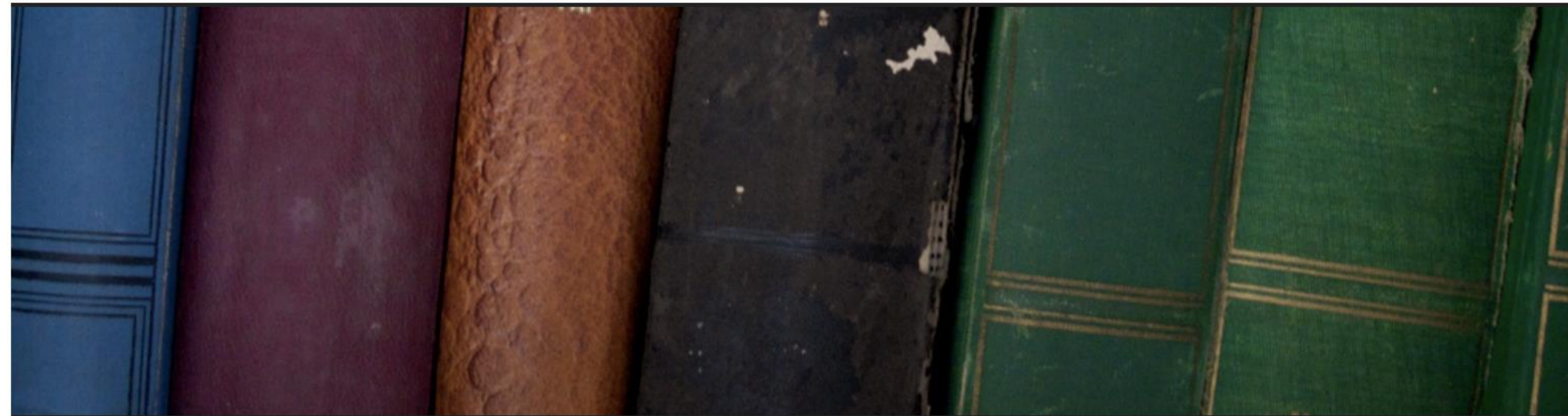
Type Safety

What is type safety? - The PL

pl-enthusiast.net/2014/...

The Programming Languages Enthusiast

HOME   ABOUT THE PL ENTHUSIAST



← Program verification in the undergraduate CS curriculum (Part II)

Spotlight: Cindy Rubio Gonzalez →

BY MICHAEL HICKS | AUGUST 5, 2014 · 7:15 AM

↓ Jump to Comments

What is type safety?

In response to my previous post defining memory safety (for C), one commenter suggested it would be nice to have a post explaining type safety. Type safety is pretty well understood, but it's still not something you can easily pin down. In particular, when someone says, "Java is a type-safe language," what do they mean, exactly? Are all type-safe languages "the same" in some way? What is type safety getting you, for particular languages, and in general?

In fact, what type safety means depends on language type system's definition. In the simplest case, type safety ensures that program behaviors are well defined. More generally, as I discuss in this post, a language's type system can be a

Search

Recent Posts

- How to Write a Grad School Personal Statement
- BullFrog: Online Schema Migration, On Demand
- Increasing the Impact of PL Research
- "What is PL Research?" The Talk
- How to Write a Conference Talk

Subscribe to Blog via Email

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

Email Address

http://www.pl-enthusiast.net/2014/08/05/type-safety/

Type safety

- Each object is ascribed a **type** (`int`, pointer to `int`, pointer to function, …), and
- Operations on the object are always *compatible* with the object's type
  - Type safe programs do not "go wrong" at run-time
- **Type safety** is **stronger** than memory safety

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
cmp("hello","bye"); // crash!
```

Memory safe,
but not type safe

Dynamically typed languages

- **Dynamically typed languages**, like Ruby and Python, which do not require declarations that identify types, can be viewed as **type safe** as well
- Each **object** has **one type: Dynamic**
  - Each operation on a Dynamic object is permitted, but *may be unimplemented*
  - In this case, it *throws an exception*

Well-defined
(but unfortunate)

Enforce invariants

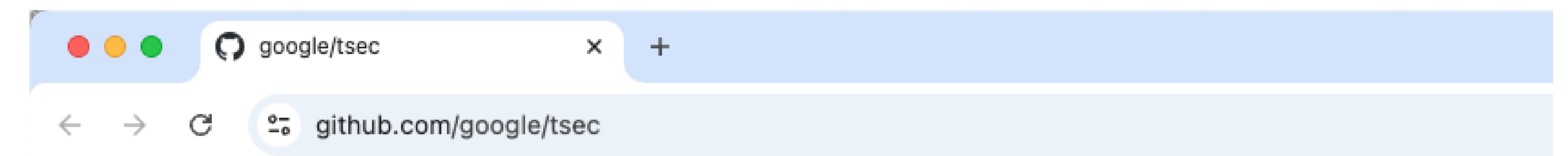
- Types really show their strength by **enforcing invariants** in the program
- Notable here is the enforcement of **abstract types**, which characterize modules that keep their **representation hidden** from clients
  - As such, we can reason more confidently about their **isolation** from the rest of the program

Type safety ensures safe coding

```
// Navigate window to a new URL.
window.location.href = 'https://example.com';
// Append new HTML markup to the current document.
document.write(html);

// Signature of a safe setter for Location#href.
// Values assigned to the sink must have the
// `SafeUrl` type.
declare function safeSetLocationHref(
  loc: Location, url: SafeUrl): void;

// Signature of a safe version of Document#write.
// Values appended must have the `SafeHtml` type.
declare function safeDocumentWrite(
  doc: Document, html: SafeHtml): void;

// Navigate window to a safe URL built from literal
safeSetLocationHref(
  window.location,
  SafeUrl.fromLiteral('https://example.com'),
);
// Append sanitized HTML markup to the current
// document.
safeDocumentWrite(document, SafeHtml.sanitize(html));
```

Type error! Requires SafeURL

Type error! Requires SafeHTML



README Code of conduct Contributing Apache-2.0 license Security

tsec - Extended TypeScript compiler checking Trusted Types compatibility

This is not an officially supported Google product.

Ok! A literal is a SafeURL

tsec supports most compilation flags as tsc does. For code pattern patterns that is potentially incompatible with Trusted Types, tsec emits compilation errors.

Ok! Surely-sanitized SafeHTML

checks ban using dangerous DOM sink APIs with plain string values. Any violation of the checks can hinder Trusted Types adoption either directly or indirectly. To fix the violations, you should construct Trusted Types values to feed into these sinks. At the moment, tsec covers most of the Trusted Types sinks that are enforced by the browser. See here for the complete list of available checks. We will be adding the missing ones soon.

Exotic types for security

- **Type-enforced invariants can relate directly to security properties**
  - By expressing stronger invariants about data's privacy and integrity, which the type checker then enforces
- **Example**: **Java with Information Flow (JIF)**

```
int{Alice→Bob} x;
int{Alice→Bob, Chuck} y;
x = y; //OK: policy on x is stronger
y = x; //BAD: policy on y is not
       //as strong as x
```

Types have *security labels*

Labels define what information flows allowed

http://www.cs.cornell.edu/jif

Why memory and type safety matter

Other defensive strategies

Against the possibility of memory safety exploits

Reduce chances of introducing a bug

- Secure coding practices
- Advanced code review and testing
  - E.g., program analysis, penetrating testing (fuzzing)

Make a bug harder to exploit

- Examine necessary steps for exploitation, make one or more of them difficult, or impossible

Limit the damage of exploitation

- Examine necessary steps for exploitation, make one or more of them difficult, or impossible

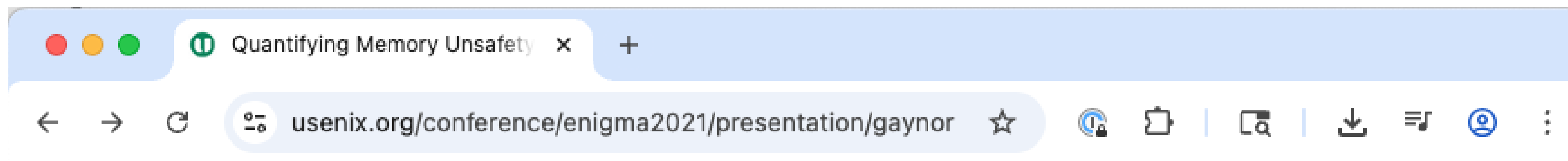
Challenge exploitability

- Stack canaries
- Address-space layout randomization (ASLR)
- “write xor execute” (W⊕X)
- Control-flow integrity (CFI)

Are these defenses working?

Limit the damage

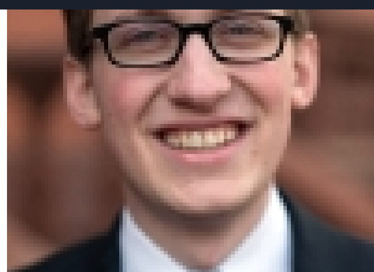
- Process-level isolation
- Within-process compartments (eg., RLbox)



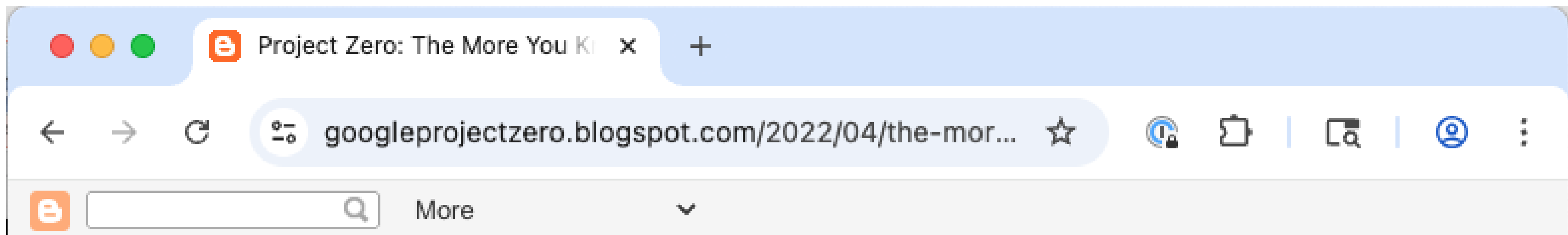
QUANTIFYING MEMORY UNSAFETY AND REACTIONS TO IT

Wednesday, February 03, 2021 - 9:20 am–9:50 am

- **Chrome**: 70% of high/critical vulnerabilities are memory unsafety
- **Firefox**: 72% of vulnerabilities in 2019 are memory unsafety
- **0days**: 81% of in the wild 0days (P0 dataset) are memory unsafey
- **Microsoft**: 70% of all MSRC tracked vulnerabilities are memory unsafety
- **Ubuntu**: 65% of kernel CVEs in USNs in a 6-month sample are memory unsafety
- **Android**: More than 65% of high/critical vulnerabilities are memory unsafety
- **macOS**: 71.5% of Mojave CVEs are due to memory unsafety



an engineer at Mozilla and the United States Digital Service. Alex has a long history of contribution in open source, from building a JIT'd Ruby VM to serving on the Board of Directors of the Python Software Foundation. Alex lives in Washington, D.C.



Project Zero

News and updates from the Project Zero team at Google

Tuesday, April 19, 2022

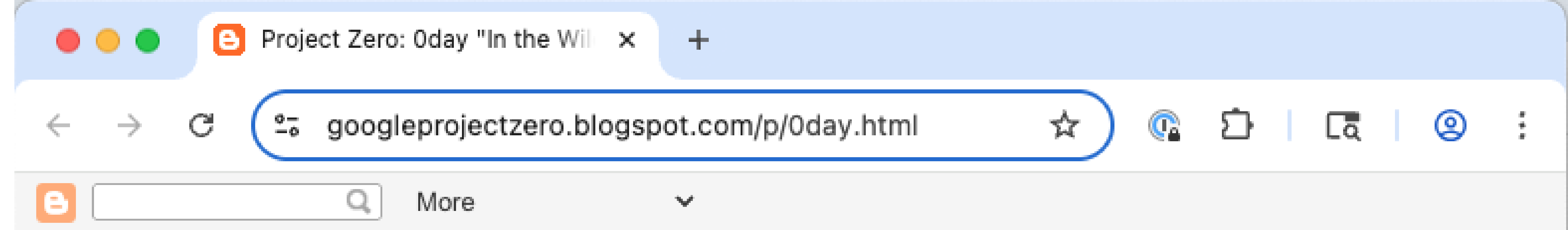
The More You Know, The More You Know You Don't Know

A Year in Review of 0-days Used In-the-Wild in 2021

Posted by Maddie Stone, Google Project Zero

This is our third annual year in review of 0-days exploited in-the-wild [2020, 2019]. Each year we've looked back at all of the detected and disclosed in-the-wild 0-days as a group and synthesized what we think the trends and takeaways are. The goal of this report is not to detail each individual exploit, but instead to analyze the exploits from the year as a group, looking for trends, gaps, lessons learned, successes, etc. If you're interested in the analysis of individual exploits, please check out our root cause analysis repository.

We perform and share this analysis in order to **make 0-day hard**. We want it to be more costly, more



Project Zero

News and updates from the Project Zero team at Google

0day "In the Wild"

Posted by Ben Hawkes, Project Zero (2019-05-15)

Project Zero's team mission is to "make zero-day hard", i.e. to make it more costly to discover and exploit security vulnerabilities. We primarily achieve this by performing our own security research, but at times we also study external instances of zero-day exploits that were discovered "in the wild". These cases provide an interesting glimpse into real-world attacker behavior and capabilities, in a way that nicely augments the insights we gain from our own research.

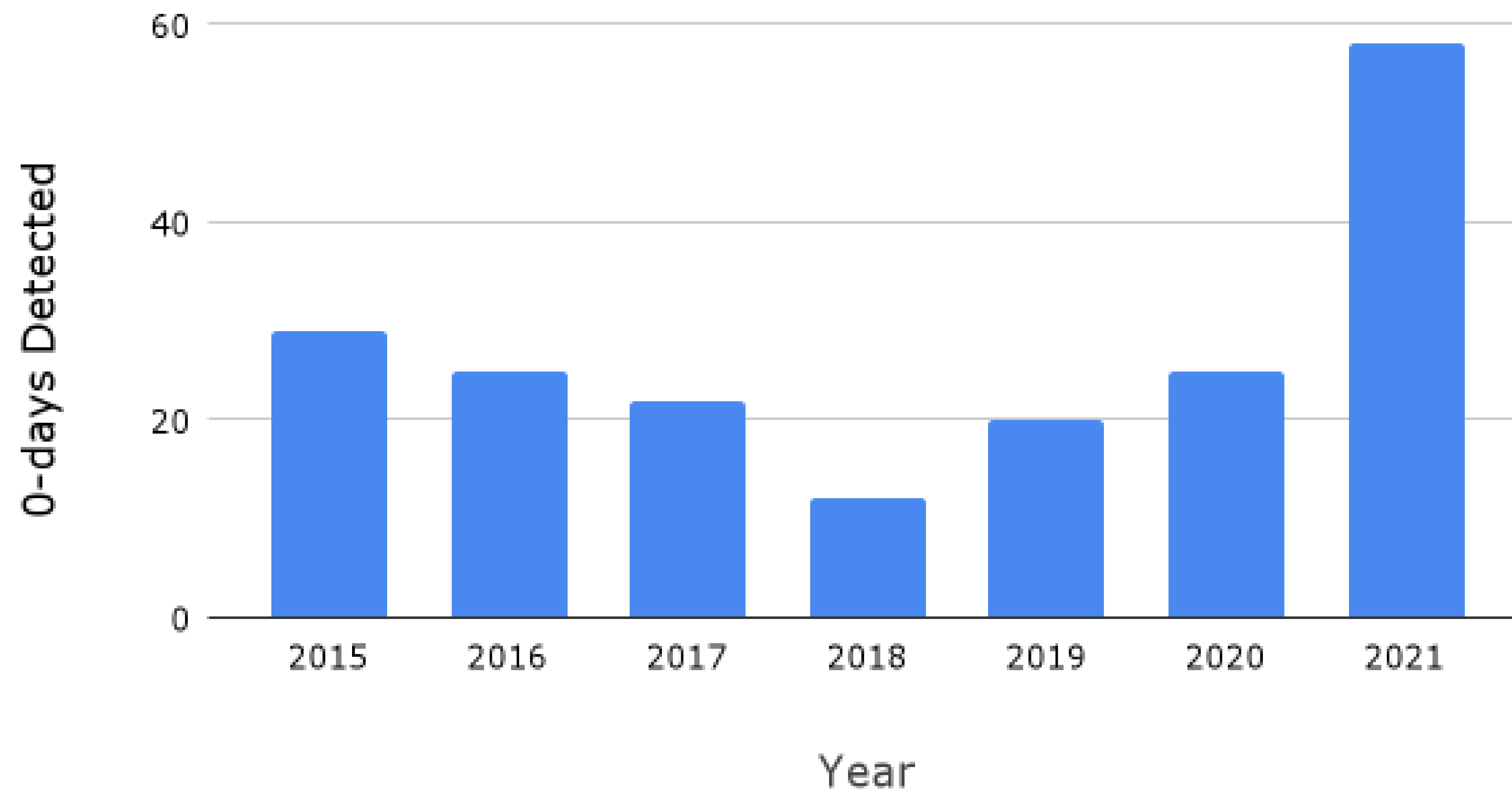
Today, we're sharing our tracking spreadsheet for publicly known cases of detected zero-day exploits, in the hope that this can be a useful community resource:

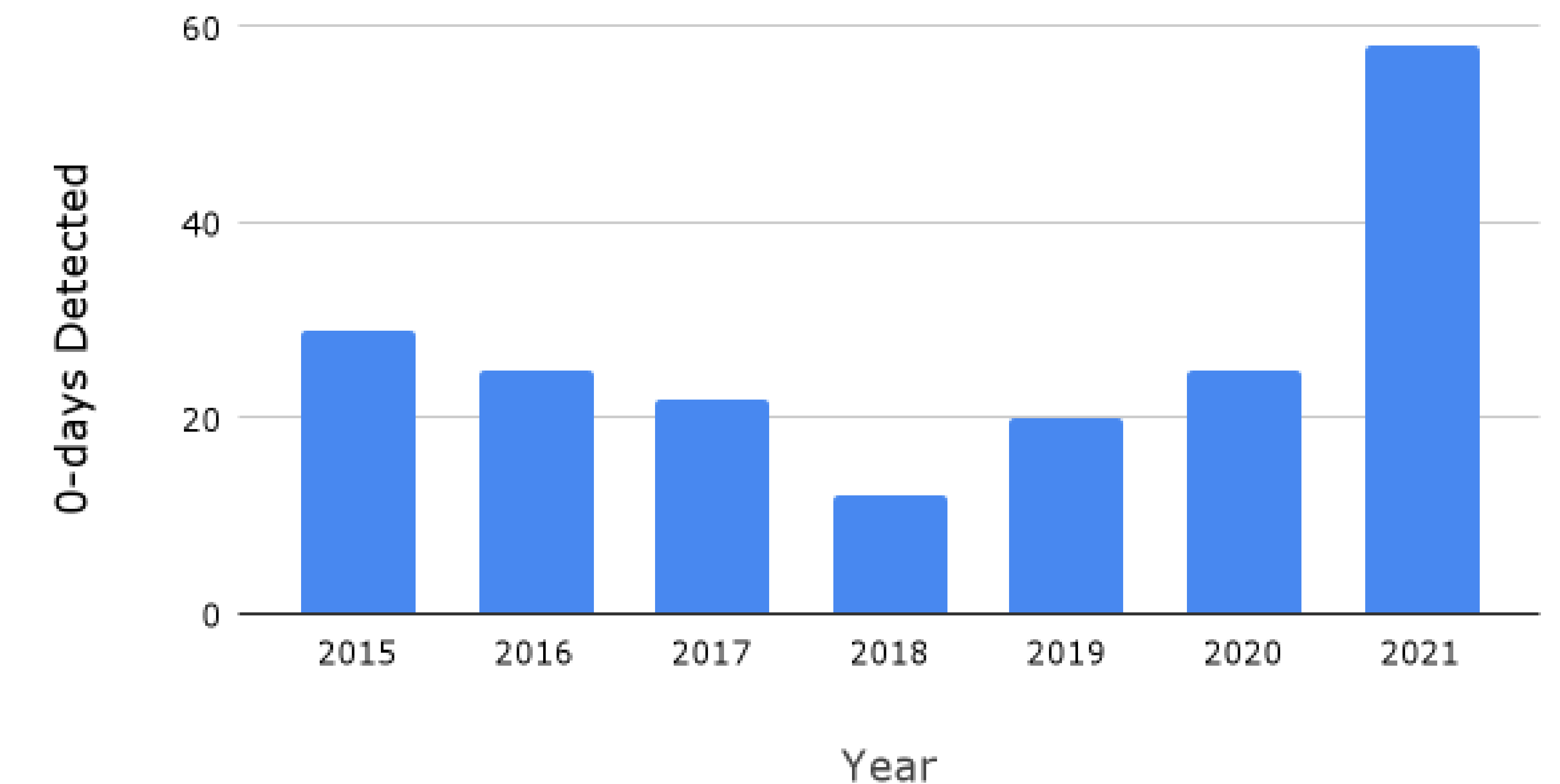
Spreadsheet link: 0day "In the Wild"

This data is collected from a range of public sources. We include relevant links to third-party analysis and attribution, but we do this only for your information; their inclusion does not mean we endorse or validate the content there. The data described in the spreadsheet is nothing new, but we think that collecting it together in one place is useful. For example, it shows that:

- On average, a new "in the wild" exploit is discovered every 17 days (but in practice these often clump together in exploit chains that are all discovered on the same date);
- Across all vendors, it takes 15 days on average to patch a vulnerability that is being used in active attacks;
- A detailed technical analysis on the root-cause of the vulnerability is published for 86% of listed CVEs;
- Memory corruption issues are the root-cause of 68% of listed CVEs.

We also think that this data poses an interesting question: *what is the detection rate of 0day exploits?* In other words, at what rate are 0day exploits being used in attacks *without* being detected? This is a key "unknown parameter" in security, and how you model it will greatly inform your views, plans, and priorities as a defender.


In-the-Wild 0-days Detected vs. Year
60
40
20
0
0-days Detected
2015
2016
2017
2018
2019
2020
2021
Year



New Year, Old Techniques

We had a record number of "data points" in 2021 to understand how attackers are actually using 0-day exploits. A bit surprising to us though, out of all those data points, there was nothing new amongst all th data. 0-day exploits are considered one of the most advanced attack methods an actor can use, so it w be easy to conclude that attackers must be using special tricks and attack surfaces. But instead, the 0-days we saw in 2021 generally followed the same bug patterns, attack surfaces, and exploit "shapes" previously seen in public research. Once "0-day is hard", we'd expect that to be successful, attackers would have to find new bug classes of vulnerabilities in new attack surfaces using never before seen exploitation methods. In general, that wasn't what the data showed us this year. With two exceptions (described below in the iOS section) out of the 58, everything we saw was pretty "meh" or standard.

Out of the 58 in-the-wild 0-days for the year, 39, or 67% were memory corruption vulnerabilities. Memory corruption vulnerabilities have been the standard for attacking software for the last few decades and it's still how attackers are having success. Out of these memory corruption vulnerabilities, the majority also stuck with very popular and well-known bug classes:

- 17 use-after-free
- 6 out-of-bounds read & write
- 4 buffer overflow
- 4 integer overflow

Out of the 58 in-the-wild 0-days for the year, 39, or ***67% were memory corruption vulnerabilities.***

MITRE Top 25 CWEs

2025 CWE Top 25

Spatial vulnerabilities (buffer overflow)

Temporal memory safety vulnerabilities

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2024
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	60.38	7	0
2	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	28.72	4	+1
3	CWE-352	Cross-Site Request Forgery (CSRF)	13.64	0	+1
4	CWE-862	Missing Authorization	13.28	0	+5
5	CWE-787	Out-of-bounds Write	12.68	12	-3
6	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	8.99	10	-1
7	CWE-416	Use After Free	8.47	14	+1
8	CWE-125	Out-of-bounds Read	7.88	3	-2
9	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	7.85	20	-2
10	CWE-94	Improper Control of Generation of Code ('Code Injection')	7.57	7	+1
11	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	6.96	0	N/A
12	CWE-434	Unrestricted Upload of File with Dangerous Type	6.87	4	-2
13	CWE-476	NULL Pointer Dereference	6.41	0	+8
14	CWE-121	Stack-based Buffer Overflow	5.75	4	N/A
15	CWE-502	Deserialization of Untrusted Data	5.23	11	+1
16	CWE-122	Heap-based Buffer Overflow	5.21	6	N/A
17	CWE-863	Incorrect Authorization	4.14	4	+1
18	CWE-20	Improper Input Validation	4.09	2	-6
19	CWE-284	Improper Access Control	4.07	1	N/A
20	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	4.01	1	-3
21	CWE-306	Missing Authentication for Critical Function	3.47	11	+4
22	CWE-918	Server-Side Request Forgery (SSRF)	3.36	0	-3
23	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	3.15	2	-10
24	CWE-639	Authorization Bypass Through User-Controlled Key	2.62	0	+6
25	CWE-770	Allocation of Resources Without Limits or Throttling	2.54	0	+1

Google rushes Chrome updat ×

Ask Gemini

theregister.com/2026/03/13/google_zeroday_chrome_upd...

SIGN IN / UP

The Register®

PATCHES

8

Google rushes Chrome update fixing two zero-days already under attack

Skia graphics lib and V8 JavaScript engine brings browser's tally of actively exploited bugs to three in 2026

Carly Page

Fri 13 Mar 2026 // 11:25 UTC

Google has pushed out an emergency Chrome update to fix two previously unknown vulnerabilities that attackers were already exploiting before the patches landed.

The bugs, tracked as CVE-2026-3909 and CVE-2026-3910, affect core components of the browser and have prompted the usual warning from Google that technical details will remain under wraps until most users have updated.

"Access to bug details and links may be kept restricted until a majority of users are updated with a fix. We will also retain restrictions if the bug exists in a third-party library that other projects similarly depend on, but haven't yet fixed," the company said.

CVE-2026-3909 is an out-of-bounds write flaw in Skia, the graphics library Chrome uses to render web content and parts of its user interface. Memory corruption bugs like this can sometimes be abused by attackers to crash applications or run their own code if successfully exploited.

March 13, 2026

CVE-2026-3909 is an out-of-bounds write flaw in Skia, the graphics library Chrome uses to render web content and parts of the user interface

Why Has Memory Safety Been So Hard?

The Performance objection

Historic argument: memory safety costs too much performance

- Garbage collection pauses (Java, Go)
- Runtime bounds-check overhead
- Fat-pointer overhead in C retrofits

The C/C++ position: direct memory control is essential for systems code (kernels, game engines, embedded)

The Legacy Code problem

- An enormous amount of critical software is C/C++: Linux kernel, network stacks, browser engines, embedded firmware
- Complete rewrites are **expensive, risky**, and can introduce new bugs
- Example: *gitoxide* (Rust reimplementation of git) — multi-year effort, still working toward feature parity

The Interoperability problem

- Memory-safe and memory-unsafe code must coexist
- FFI (Foreign Function Interface) calls cross language boundaries
- A Rust program calling C via FFI requires `unsafe` blocks
- The safety boundary is only as strong as its unsafe perimeter

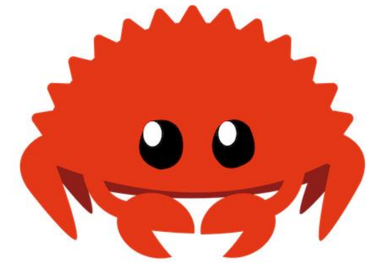
Ergonomics and ecosystem maturity

- Early memory-safe languages (Java, Go) weren't suited for systems programming
- Claim: Rust has a **steep learning curve** (the borrow checker takes time)
- Tooling, libraries, and developer familiarity take years to mature

The “Just write better code” fallacy

“Skilled programmers should not need safety guardrails”

- No historical precedent for a step-function improvement in human error rates from deciding to “try harder”
- Improvements come from **better tools and processes**
- We do not rely on pilots to “just be more careful” — we build checklists, automation, and redundant systems



Is Rust the answer?

- Performance: Rust provides low-level control and good performance
- Legacy code: There are paths between "accept unsafety forever" and "rewrite everything immediately"
- Interoperability: Rust tools help with this
- Ergonomics and maturity: Rust tools, ecosystem are excellent, and LLMs provide further help

Type- and Memory-Safety in Rust

The core insight

Make unsafe programs **inexpressible in the type system**

- Bounds checks often removed (access proved safe)
- Zero runtime overhead for ownership/borrowing
- No garbage collector

Concurrency safety too!

Ownership: Every value has one owner

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;  // ownership moves to s2

    // println!("{}", s1);  // COMPILE ERROR:
    //   value borrowed here after move

    println!("{}", s2);  // OK — s2 owns the data
}
```

Ownership: Automatic deallocation (RAII)

```
fn main() {
    {
        let s = String::from("hello");
        println!("{}", s);
    } // s is dropped here — memory freed automatically

    // println!("{}", s);  // COMPILE ERROR:
    //   not found in this scope
}
```

Manual `drop()` ok in some cases, as checked by compiler

Borrowing: The exclusivity invariant

At any point, you may have **either:**

- Any number of **immutable references** (`&T`), OR
- Exactly **one mutable reference** (`&mut T`)

Never both simultaneously

Borrowing in action

```
fn main() {
    let mut data = vec![1, 2, 3];

    let r1 = &data;       // immutable borrow — OK
    let r2 = &data;       // another immutable borrow — OK
    println!("{:?} {:?}", r1, r2);

    let r3 = &mut data;   // mutable borrow — OK (r1,r2 done)
    r3.push(4);
    println!("{:?}", r3);
}
```

Borrowing prevents Iterator invalidation

```rust
fn main() {
    let mut numbers = vec![1, 2, 3];

    for n in &numbers {
        // numbers.push(42);  // COMPILE ERROR:
        //   cannot borrow `numbers` as mutable
        //   because it is also borrowed as immutable
        println!("{}", n);
    }
}
```

Lifetimes: No dangling references

```rust
// This does NOT compile:
fn dangling() -> &String {
    let s = String::from("hello");
    &s    // ERROR: s is dropped here,
}         //   reference would dangle!

// The safe version — return the owned value:
fn not_dangling() -> String {
    let s = String::from("hello");
    s    // ownership moves to caller
}
```

Spatial safety: Bounds checking

```rust
fn main() {
    let data = vec![10, 20, 30, 40, 50];

    // Runtime panic — NOT undefined behavior:
    // println!("{}", data[10]);
    //   panicked at 'index out of bounds:
    //   the len is 5 but the index is 10'

    // Safe alternative with .get():
    match data.get(10) {
        Some(val) => println!("{}", val),
        None      => println!("Out of bounds!"),
    }
}
```

The `unsafe` escape hatch

`unsafe` allows:

- Dereferencing raw pointers
- Calling unsafe functions (e.g., FFI)
- Accessing mutable globals
- Implementing unsafe traits

`unsafe` does **not** disable the borrow checker for surrounding safe code

The `unsafe` escape hatch

```rust
let mut data = vec![1, 2, 3, 4, 5];
// split_at_mut needs unsafe internally because
// the borrow checker can't verify non-overlapping
// mutable slices — but it presents a safe API:
let (left, right) = data.split_at_mut(3);
left[0] = 10;
right[0] = 40;
println!("After split_at_mut: {:?}", data);
// prints [10, 2, 3, 40, 5]
```

Rust vs. prior objections: Scorecard

Objection	Before Rust	Rust
GC pauses	Java, Go	No GC; RAII deallocation
Runtime overhead	Bounds-checked C	Elided where proven
Performance	“Use C/C++”	Comparable to C/C++
Systems programming	C only	Linux kernel, Android
C interop	Hard (Java)	FFI with explicit `unsafe` boundary

Making Memory Safety Mainstream

Four steps of systemic change

Google's **four-generation framework** [Vander Stoep & Rebert, Google Security Blog, Sept 2024]:

1. **Reactive patching** — fix after exploitation
2. **Proactive mitigations** — ASLR, canaries, CFI
3. **Proactive discovery** — fuzzing, sanitizers
4. **Safe-by-design languages** — prevent the bug class entirely

Safe Coding

TLP:CLEAR

The Case for Memory Safe Roadmaps

Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously

Publication: December 2023

United States Cybersecurity and Infrastructure Security Agency
United States National Security Agency
United States Federal Bureau of Investigation
Australian Signals Directorate's Australian Cyber Security Centre
Canadian Centre for Cyber Security
United Kingdom National Cyber Security Centre
New Zealand National Cyber Security Centre
Computer Emergency Response Team New Zealand

This document is marked TLP:CLEAR. Disclosure is not limited. Sources may use TLP:CLEAR when information carries minimal or no foreseeable risk of misuse, in accordance with applicable rules and procedures for public release. Subject to standard copyright rules, TLP:CLEAR information may be distributed without restriction. For more information on the Traffic Light Protocol, see cisa.gov/tlp.

TLP:CLEAR

December 2023

TLP:CLEAR

Exploring Memory Safety in Critical Open Source Projects

Publication: June 26, 2024

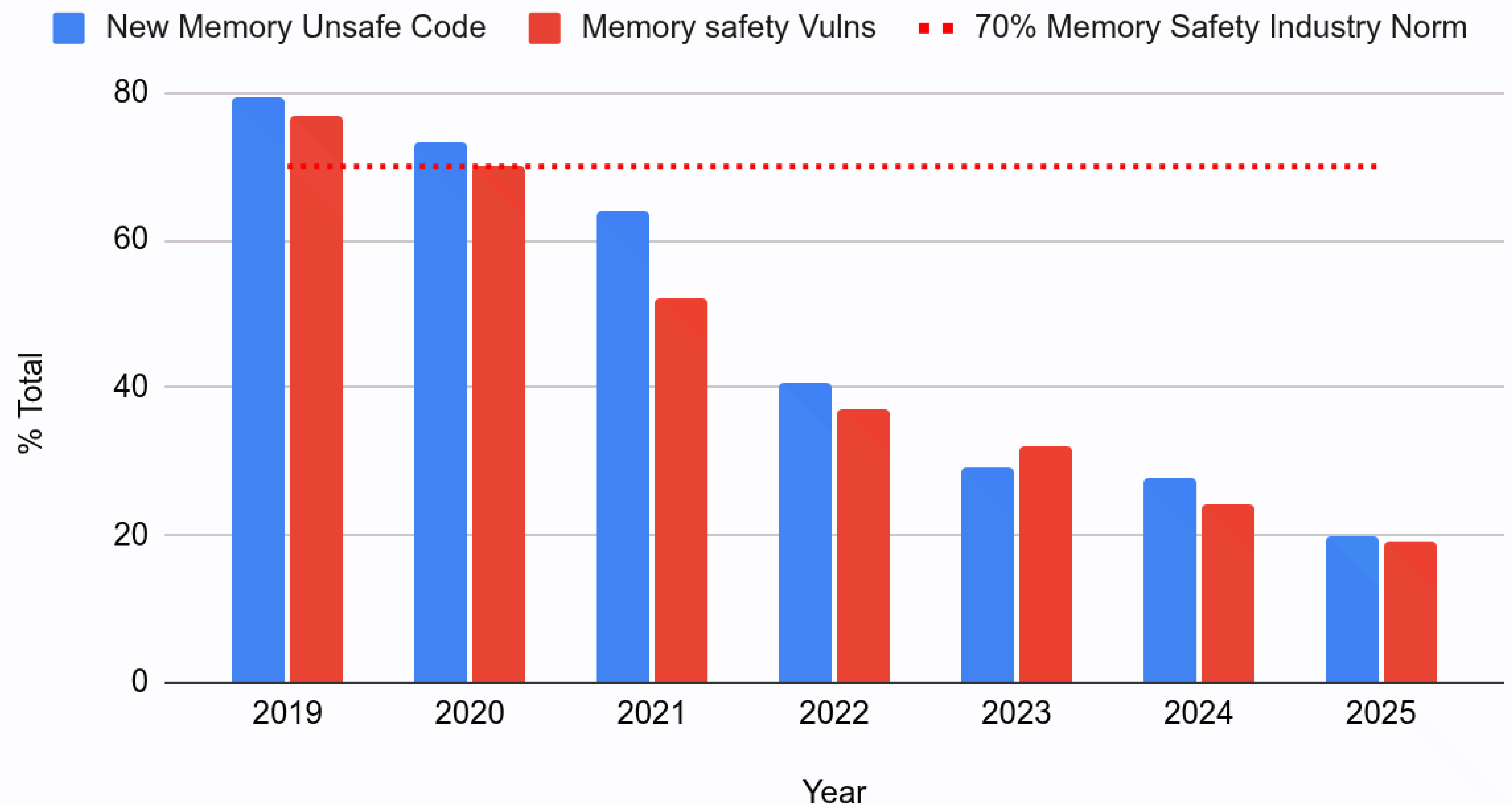
Cybersecurity and Infrastructure Security Agency (CISA)
Federal Bureau of Investigation (FBI)
Australian Signals Directorate's (ASD's) Australian Cyber Security Centre (ACSC)
Canadian Centre for Cyber Security (CCCS)

This document is marked TLP:CLEAR. Disclosure is not limited. Sources may use TLP:CLEAR when information carries minimal or no foreseeable risk of misuse, in accordance with applicable rules and procedures for public release. Subject to standard copyright rules, TLP:CLEAR information may be distributed without restriction. For more information on the Traffic Light Protocol, see cisa.gov/tlp.

June 2024

Android: Showing it can work

- Android team began using Rust for **new code** in 2019
- **No mass rewrite** of existing C/C++
- Memory safety CVEs: **76% → 20%** (74% reduction)

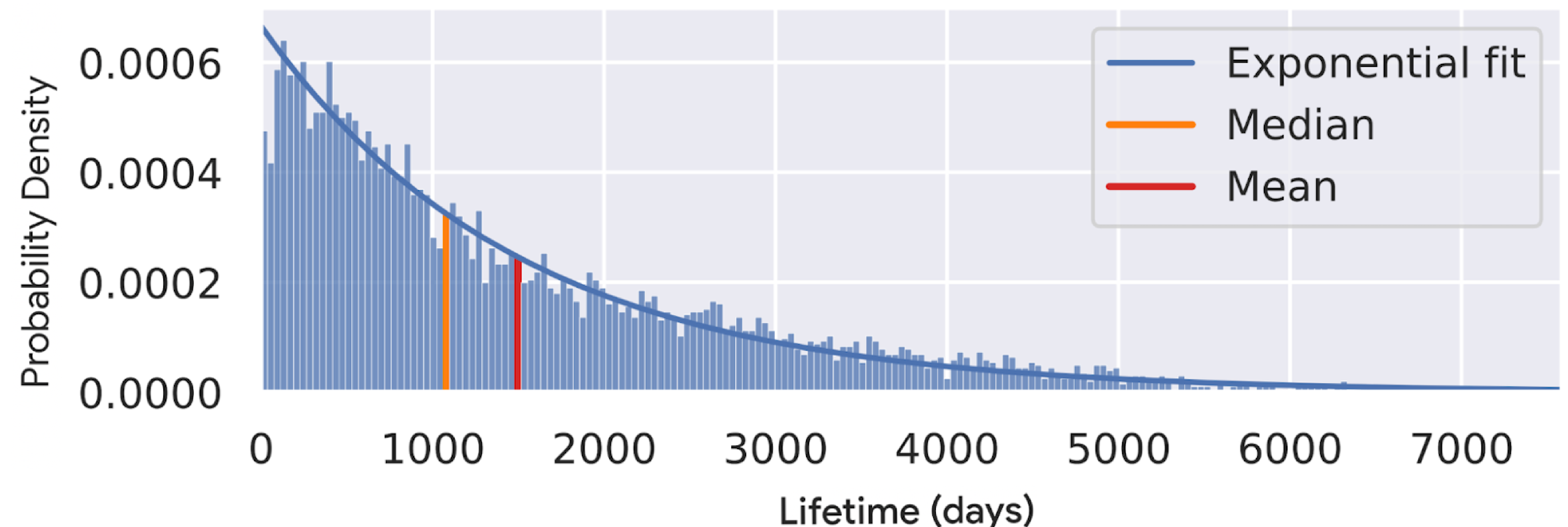


https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html

The bug half-life insight

“Bugs have a half-life — older code has had its bugs found and fixed”

- New code has the **highest vulnerability density**
- Writing new code safely has **compounding benefit**



- You don’t need to rewrite everything — just stop introducing new bugs

https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos

Rust in the Linux kernel

October 2022

reddit.com/r/rust/comments/zjczmq/linux_61_released_...

r/rust · 3y ago
[deleted]

Linux 6.1 Released With Initial Rust Code

```
author      Linus Torvalds <torvalds@linux-foundation.org>  2022-12-11 14:15:18 -0800
committer   Linus Torvalds <torvalds@linux-foundation.org>  2022-12-11 14:15:18 -0800
commit      830b3c68c1fb1e9176028d02ef86f3cf76aa2476 (patch)
tree        c7c1b7db9ced3eba518cfc1f711e9d89f73f8667
parent      d92b86f672a42d9d74a24a63a1e59793c4116830 (diff)
download    linux-830b3c68c1fb1e9176028d02ef86f3cf76aa2476.tar.gz
```

Linux 6.1 HEAD v6.1 master

Diffstat
-rw-r--r-- Makefile 2

1 files changed, 1 insertions, 1 deletions

```
diff --git a/Makefile b/Makefile
index 0992f827888dd..997b6772
--- a/Makefile
+++ b/Makefile
@@ -2,7 +2,7 @@
 VERSION = 6
 PATCHLEVEL = 1
 SUBLEVEL = 0
-EXTRAVERSION = -rc8
+EXTRAVERSION =
 NAME = Hurr durr I'ma ninja
```

phoronix.com Open

December 2025

lwn.net/Articles/1049831/

LWN.net News from the source

Content: Weekly Edition, Archives, Search, Kernel, Security, Events calendar, Unread comments, LWN FAQ, Write for us

Edition: Return to the Briefs page

The (successful) end of the kernel Rust experiment

[Posted December 10, 2025 by corbet]

The topic of the Rust experiment was just discussed at the annual Maintainers Summit. The consensus among the assembled developers is that Rust in the kernel is no longer experimental — it is now a core part of the kernel and is here to stay. So the "experimental" tag will be coming off. Congratulations are in order for all of the Rust for Linux team.

(Stay tuned for details in our Maintainers Summit coverage.)

Log in to post comments

[–] Nice
Posted Dec 10, 2025 4:25 UTC (Wed) by ktkaffee (subscriber, #112877) [Link] (19 responses)

You got me for a second

[–] Nice
Posted Dec 10, 2025 4:45 UTC (Wed) by josh (subscriber, #17465) [Link] (13 responses)

[–] Nice
Posted Dec 10, 2025 5:24 UTC (Wed) by josh (subscriber, #17465) [Link]

Challenge: kernel patterns (shared mutable state, cyclic structures) push against Rust's ownership model

The Great Refactor

Proposal: government-funded AI-assisted translation of critical C/C++ → Rust

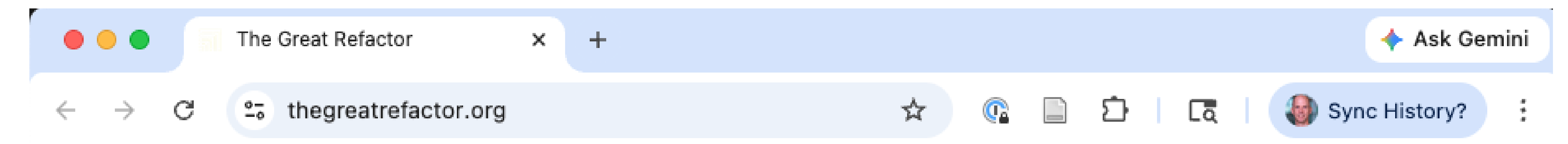
- Target: ~100 million lines of OSS code by 2030
- Estimated cost: $100 million
- Builds on DARPA's **TRACTOR** program (Translating All C To Rust)
- Priority: parsing libraries, network protocols, system utilities

The Great Refactor

Ask Gemini

thegreatrefactor.org

Sync History?



The Great Refactor is a focused nonprofit effort to re-write the world's critical code-bases into Rust, a programming language that ensures that code is memory-safe, eliminating a key class of cybersecurity vulnerabilities.

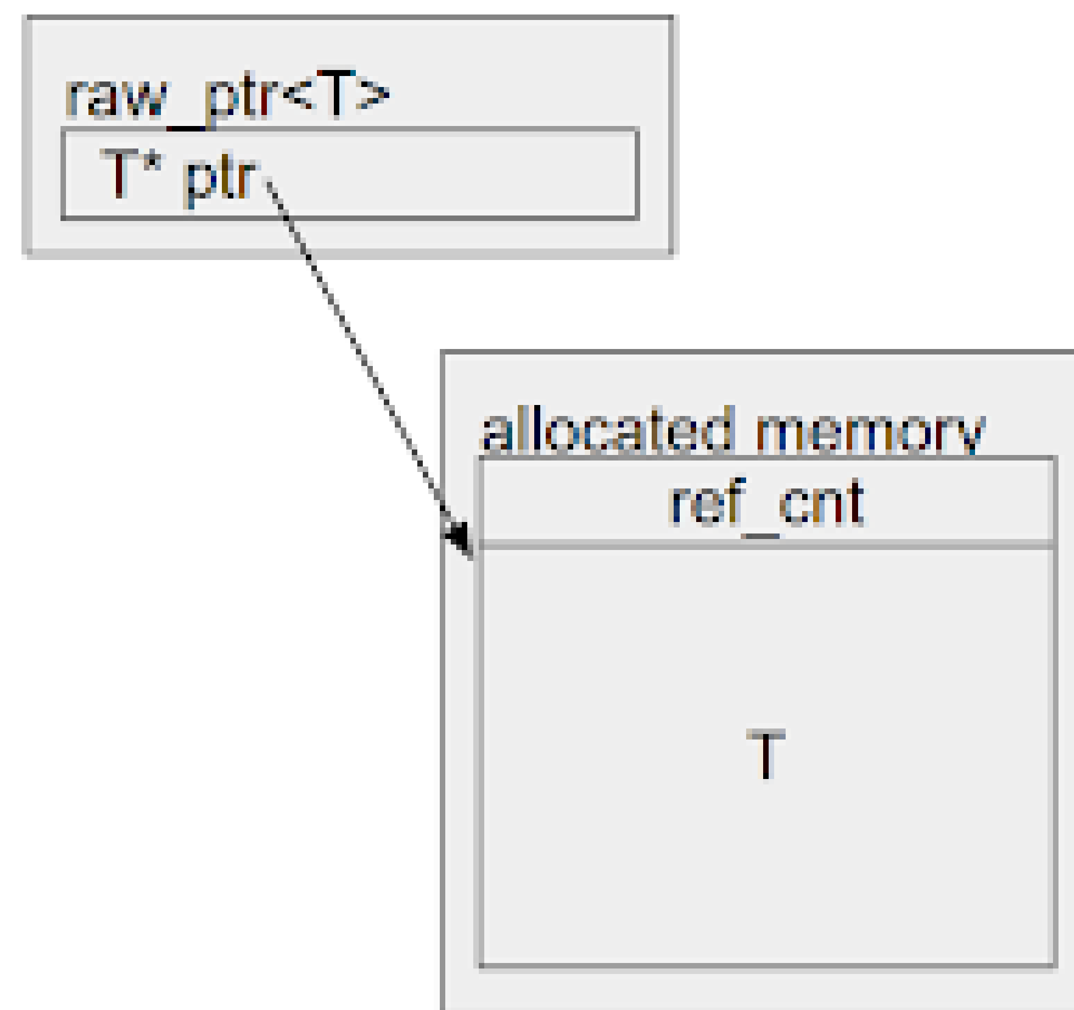
See the full idea here, and sign up for the mailing list here!

Why Memory Safety?

Memory-safety vulnerabilities account for ~70% of CVEs. While AI shows great promise for targeted patching, with translation to Rust, we can proactively eliminate an entire class of vulnerabilities. Many historic vulnerabilities that cost billions stemmed from open-source repositories with many downloads but a handful of (heroic) volunteer maintainers—Log4J and Heartbleed are classic examples.

Gradual hardening of existing code

- **MiraclePtr** (Chrome): smart pointer reducing use-after-free by >50%



Google Online Security Blog:
Ask Gemini

Use-af
Septemb

Posted by

Memory
we're co
languag
recently
Those te
likely req

Today w

It's hard,
mistake

```
class A { ... };
class B {
  B(A* a) : a_(a) {}
  void doSomething() { a_->doSomething(); }
  raw_ptr<A> a_;  // MiraclePtr
};

std::unique_ptr<A> a = std::make_unique<A>();
std::unique_ptr<B> b = std::make_unique<B>(a.get());
…
a = nullptr;  // The free is delayed because the MiraclePtr
              // is still pointing to the object.
b->doSomething();  // Use-after-free is neutralized.
```

assumptions about how a bit of code will work, then a later change invalidates those assumptions. Suddenly, the data isn't valid as long as the original programmer expected, and an exploitable bug results.

Gradual hardening of existing code

- **MiraclePtr** (Chrome): smart pointer reducing use-after-free by >50%
- **Bounds-checking in C++ stdlib** (Google, MSVC hardened mode)

Google Online Security Blog:
Ask Gemini
security.googleblog.com/2024/11/retrofitting-spatial-...

Retrofitting spatial safety to hundreds of millions of lines of C++

November 15, 2024

Posted by Alex Rebert and Max Shavric

Attackers regularly exploit spatia
code accesses a memory alloca
systems and sensitive data. The
users.

Based on an analysis of in-the-w
safety vulnerabilities represent 40% of in-the-wild memory safety exploits over the past

Null deref 1.5%
Thread 5.6%
Type 17.3%
Init 1.0%
Temporal 34.2%
Spatial 40.3%

Hardening Modes — libc++ d
Ask Gemini
libcxx.llvm.org/Hardening.html

libc++ documentation
HARDENING MODES

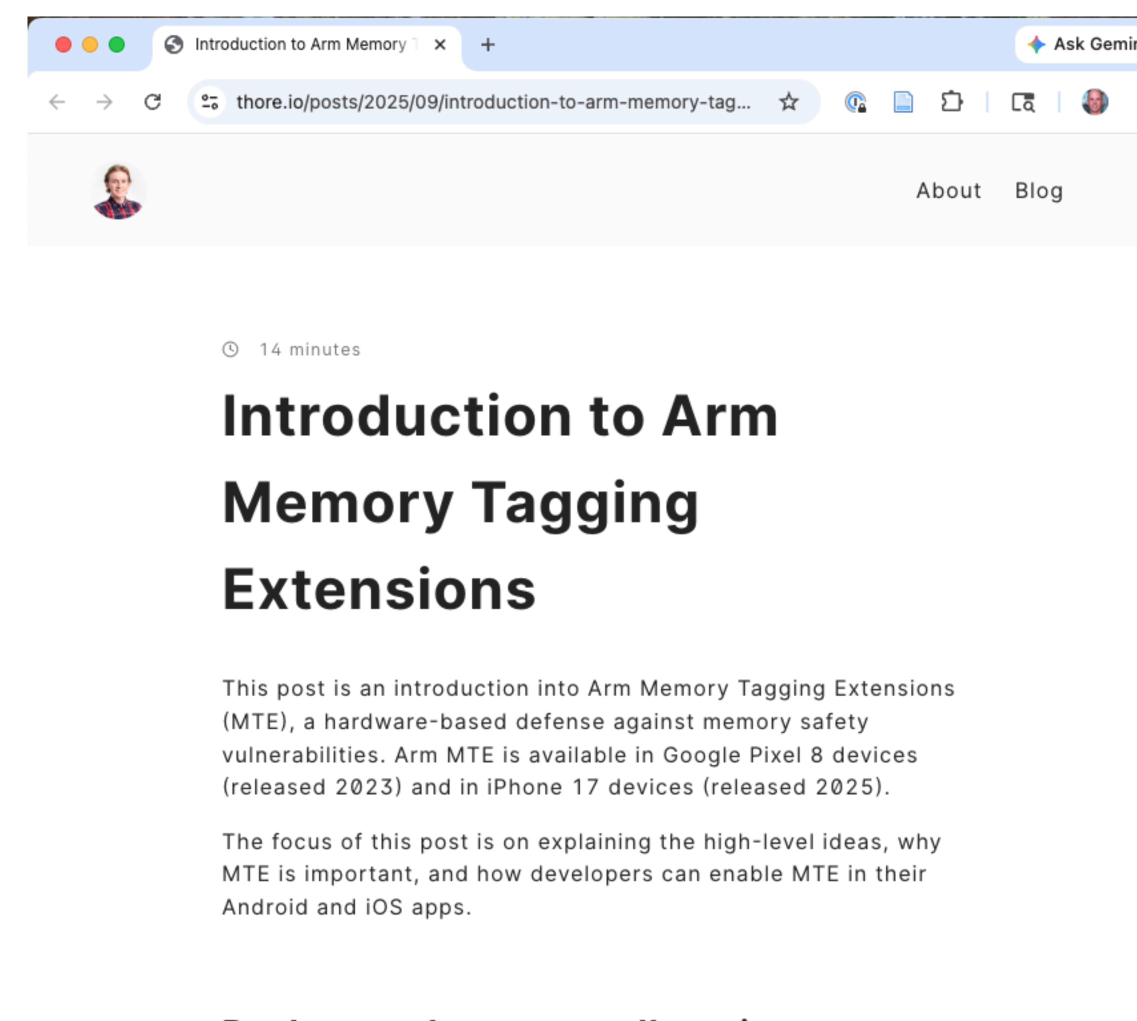
« Modules in libc++ :: Contents :: Release proce

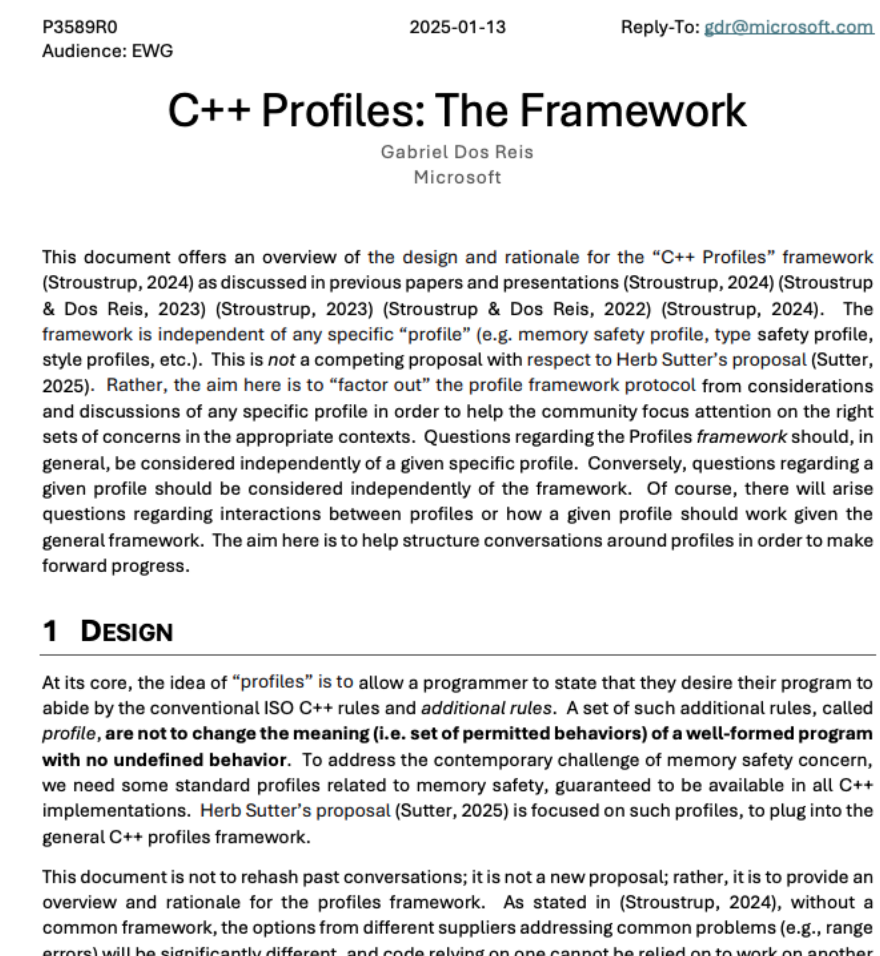
Hardening Modes

- Using hardening modes
  - Notes for users
  - Notes for vendors
- Assertion categories
- Mapping between the hardening modes and the assertion categories
- Hardening assertion failure
  - Assertion semantics
  - Notes for vendors
  - Overriding the assertion failure handler
- ABI
  - ABI options
  - ABI tags
- Hardened containers status
- Testing
- Further reading

Gradual hardening of existing code

- **MiraclePtr** (Chrome): smart pointer reducing use-after-free by >50%
- **Bounds-checking in C++ stdlib** (Google, MSVC hardened mode)
- **ARM MTE** in Android: hardware use-after-free detection
- **C++ Profiles** (C++26 proposed): compiler-enforced safe subset





P3589R0 2025-01-13 Reply-To: gdr@microsoft.com
Audience: EWG

C++ Profiles: The Framework

Gabriel Dos Reis
Microsoft

This document offers an overview of the design and rationale for the "C++ Profiles" framework (Stroustrup, 2024) as discussed in previous papers and presentations (Stroustrup, 2024) (Stroustrup & Dos Reis, 2023) (Stroustrup, 2023) (Stroustrup & Dos Reis, 2022) (Stroustrup, 2024). The framework is independent of any specific "profile" (e.g. memory safety profile, type safety profile, style profiles, etc.). This is *not* a competing proposal with respect to Herb Sutter's proposal (Sutter, 2025). Rather, the aim here is to "factor out" the profile framework protocol from considerations and discussions of any specific profile in order to help the community focus attention on the right sets of concerns in the appropriate contexts. Questions regarding the Profiles *framework* should, in general, be considered independently of a given specific profile. Conversely, questions regarding a given profile should be considered independently of the framework. Of course, there will arise questions regarding interactions between profiles or how a given profile should work given the general framework. The aim here is to help structure conversations around profiles in order to make forward progress.

1 Design

At its core, the idea of "profiles" is to allow a programmer to state that they desire their program to abide by the conventional ISO C++ rules and *additional rules*. A set of such additional rules, called *profile*, **are not to change the meaning (i.e. set of permitted behaviors) of a well-formed program with no undefined behavior**. To address the contemporary challenge of memory safety concern, we need some standard profiles related to memory safety, guaranteed to be available in all C++ implementations. Herb Sutter's proposal (Sutter, 2025) is focused on such profiles, to plug into the general C++ profiles framework.

This document is not to rehash past conversations; it is not a new proposal; rather, it is to provide an overview and rationale for the profiles framework. As stated in (Stroustrup, 2024), without a common framework, the options from different suppliers addressing common problems (e.g., range

Key Takeaways

Key Takeaways

1. Memory safety = access only *defined* memory through *legitimate* pointers
2. ~70% of security vulnerabilities are memory safety bugs
3. “Write better code” doesn’t work; better tools and languages do
4. Rust enforces safety at compile time with low runtime overhead
5. Write new code safely; harden legacy code with available tools
6. The transition is happening: Android, Linux kernel, government policy