

Multiprocessing (part 2)

CS110L
February 9th, 2022

Logistics: Syllabus for the Rest of the Class!

- Next Monday (2/14): **In-class time for project 1 work**
 - I'll be in Hewlett 101 as usual, and I'll have a Zoom link on Canvas that I'll have open so that you can join that way.
 - Come through! Talk about it! Ask questions! Collaborate! Get that participation grade!!!



*This could be you!
Next Monday in Hewlett 101
(or on Zoom)!*

Logistics: Syllabus for the Rest of the Class!

- Next Wednesday (2/16): **Multithreading I**
- 2/21: no class (president's day)
- Week 7 exercises (due 2/22; practice on stuff from Multithreading I class)
- 2/23: **Multithreading II** (some in-class coding)
- **Recorded video**: project 2 walkthrough
- 2/28: Project 2 Q&A + **in-class time for project 2 work**
- 3/2 and 3/7: somewhat tbd. Most likely: communication with **channels** and brief overview of **event-driven programming**.
- 3/9: Wrap-up
- *Note: project 2 is due on 3/11, because that's the last day I can assign something to be due, but feel free to ask for an extension. I have to submit grades on 3/22.*

Corresponding lecture recordings for multiprocessing:

- Previous multiprocessing content (2/2 and 2/7)
 - What can go wrong in fork(): second half of “Winter 2022 CS110L lecture 9” on Canvas (after generics)
 - Intro to multiprocessing in Rust + project 1 walkthrough: “Winter 2022 CS110L Lecture 10 + Project 1 Walkthrough” (on Canvas)
- Today: pipe() and how browsers use multiprocessing
 - For pipe, see “Spring 2021 CS110L lecture 11” on Canvas, starting at 34:00.
 - Previous years have also covered signal(), but we aren’t — it’s not used in CS110 anymore. (Which is good!) If you want to watch last year’s material on signal(), it’s in “Spring 2021 CS110L lecture 11” on Canvas, starting at 40:00.
 - For browser material, see “Spring 2021 CS110L lecture 12” on Canvas.

pipe(): what could go wrong?

Problems with pipes

What can you think of?

Problems with pipes

- Leaked file descriptors
- Calling close() on bad values

Example:

```
if (close(fds[1]) == -1) {  
    printf("Error closing!");  
}
```

- Use-before-pipe (i.e. use of uninitialized ints)
- Use-after-close

Potential solution

- Define a pipe type instead of using numbers!
- [Writing to a stdin pipe](#):

```
let mut child = Command::new("cat")
    .stdin(Stdio::piped())
    .stdout(Stdio::piped())
    .spawn()?;
child.stdin.as_mut().unwrap().write_all(b"Hello, world!\n")?;
let output = child.wait_with_output()?;


```

- The [os pipe crate](#) allows for creating arbitrary pipes. (The Drop trait closes the pipe.)

Problems with pipes (one more issue!)

- When a pipe “fills up”, process will block on write call until data is read out of the pipe. This can lead to deadlock, for example:
 - Say there are two pipes and two processes. Pipe #1 = process A -> process B; pipe #2 = process B -> process A
 - Process A fills up a pipe #1 — needs to stop writing until process B reads from the pipe
 - Meanwhile, process B fills up pipe #2 — needs to stop writing until process A reads
 - Process A waiting for Process B and Process B waiting for Process A
=> deadlock

What's the point?

Why are we bothering with all of this?

- In CS 110, we're spending so much time learning how to call fork and pipe and (in previous quarters) signal, and now you're telling us not to do all those things... What's up with that?
- Systems code is extremely detail oriented, and you need to have an understanding of how everything works
 - You can swap two lines and get totally different behavior
- We want you to understand how these low-level primitives work...
 - So that you can debug problems no one else can and design new types of systems
 - So that you can respect them enough to *not* use them if you don't need to

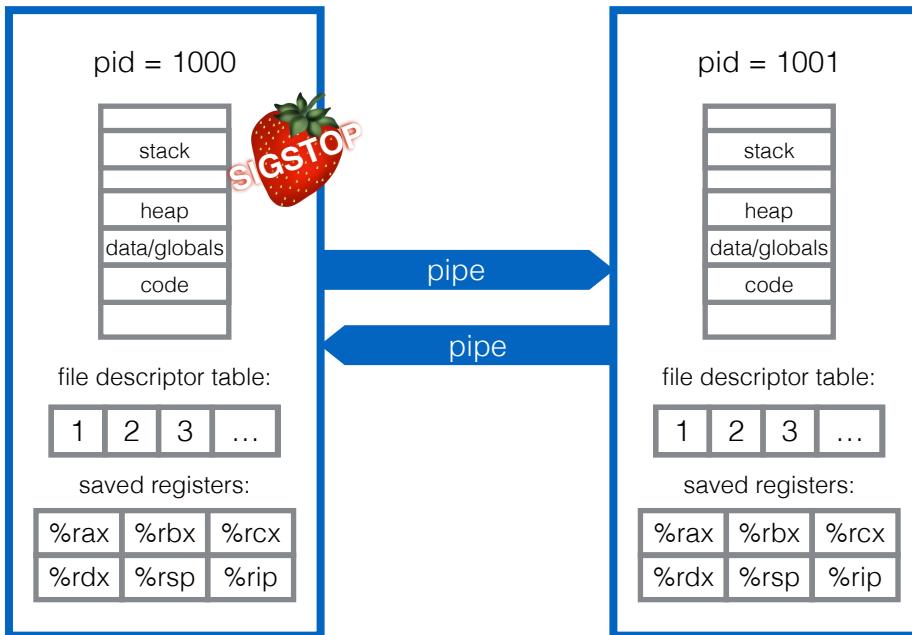
*signal() is a system call that sets up a signal handler, which is a method that is automatically invoked whenever a given signal comes in. For example, you might have a signal handler that runs when a user presses Ctrl-C (SIGINT). **We're not covering signal()**, because it's no longer used in 110! It has been replaced by teaching how to use `sigwait` for handling signals (which I think is good).*

Don't call signal()

If you're interested, check out “CS110L lecture 11” on Canvas, starting at 40:00. The corresponding slides and some lecture notes are on Ryan's website [here](#), under lecture 11.

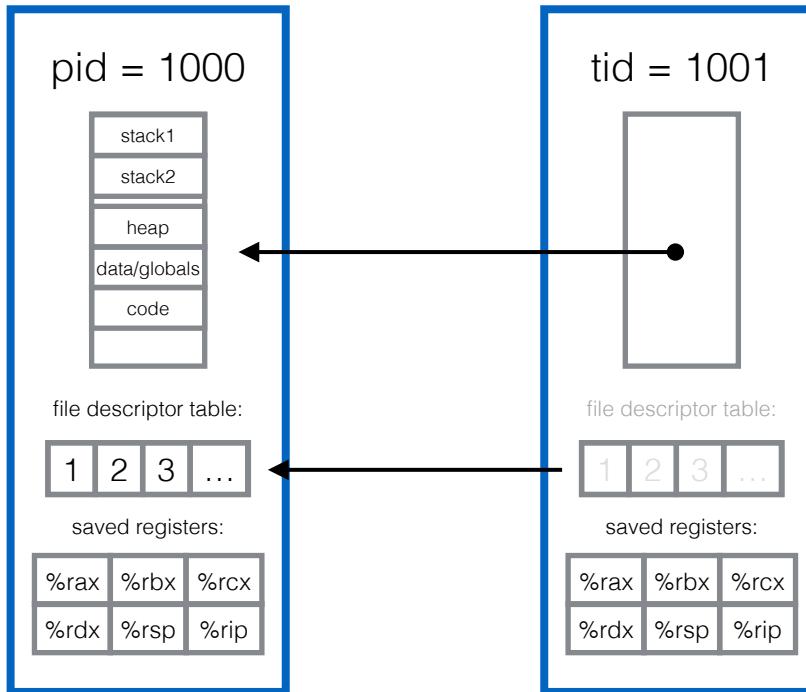
Browsers Case Study

Processes



Processes can synchronize using (generally) signals and pipes
For other approaches, see: “*inter-process communication*” ([IPC](#))

Threads



Under the hood, a thread gets its own “process control block” and is scheduled independently, but it is linked to the process that spawned it

What are the tradeoffs between threads and processes?

- If you were to make a “pro” and “con” list, what would it look like?
- If you were designing a system that supports concurrency, why might you use threads? Why might you use processes?

Considerations when designing a browser

- What do you care about?

Considerations when designing a browser

- Speed
- Resource usage (memory, battery, CPU...)
- Ease of development
- Security
- Stability

Questions:

Would threads or processes serve you better for each of these considerations?

Considerations when designing a browser

- Speed
 - Typically faster to share memory and to use lightweight synchronization primitives
 - Processes incur additional context switching overhead
- Memory, battery, and CPU usage
 - Processes use more memory
 - Processes incur additional context switching overhead
- Ease of development
 - Communication is WAY easier using threads
 - That being said, bugs caused by multithreading are extremely hard to track down
- Security, stability
 - Multiprocessing provides isolation. Multithreading does not.

Modern browsers are essentially operating systems

The screenshot shows a web browser window with the title 'Web APIs | MDN'. The URL is <https://developer.mozilla.org/en-US/docs/Web/API>. The page is titled 'Specifications' and contains a list of APIs. The APIs are grouped by letter:

- A**: Ambient Light Events
- B**: Background Tasks, Battery API, Beacon, Bluetooth API, Broadcast Channel API
- C**: CSS Counter Styles, CSS Font Loading API, CSSOM, Canvas API, Channel Messaging API, Console API, Credential Management API
- D**: DOM
- F**: Fetch API
- G**: Gamepad API, Geolocation API
- H**: HTML Drag and Drop API, High Resolution Time, History API
- I**: Image Capture API, IndexedDB, Intersection Observer API
- L**: Long Tasks API
- M**: Media Session API, Media Source Extensions, MediaStream Recording
- N**: Navigation Timing, Network Information API
- P**: Page Visibility API, Payment Request API, Performance API, Performance Timeline API, Permissions API, Pointer Events, Pointer Lock API, Proximity Events, Push API
- R**: Resize Observer API, Resource Timing API
- S**: Storage Access API, Streams
- T**: Touch Events
- U**: URL API
- V**: Vibration API
- W**: Web Animations, Web Audio API, Web Authentication API, Web Crypto API, Web Notifications, Web Storage API, Web Workers API
- WebGL**
- WebRTC**

- Storage APIs
- Concurrency APIs
- Hardware APIs (e.g. communicate with MIDI devices, GPU)
- Run assembly
- Run Windows 95:
<https://win95.ajf.me/>

<https://developer.mozilla.org/en-US/docs/Web/API>

Motivation for Chrome

It's nearly impossible to build a rendering engine that never crashes or hangs. It's also nearly impossible to build a rendering engine that is perfectly secure.

In some ways, the state of web browsers around 2006 was like that of the single-user, co-operatively multi-tasked operating systems of the past. As a misbehaving application in such an operating system could take down the entire system, so could a misbehaving web page in a web browser. All it took is one browser or plug-in bug to bring down the entire browser and all of the currently running tabs.

Modern operating systems are more robust because they put applications into separate processes that are walled off from one another. A crash in one application generally does not impair other applications or the integrity of the operating system, and each user's access to other users' data is restricted.

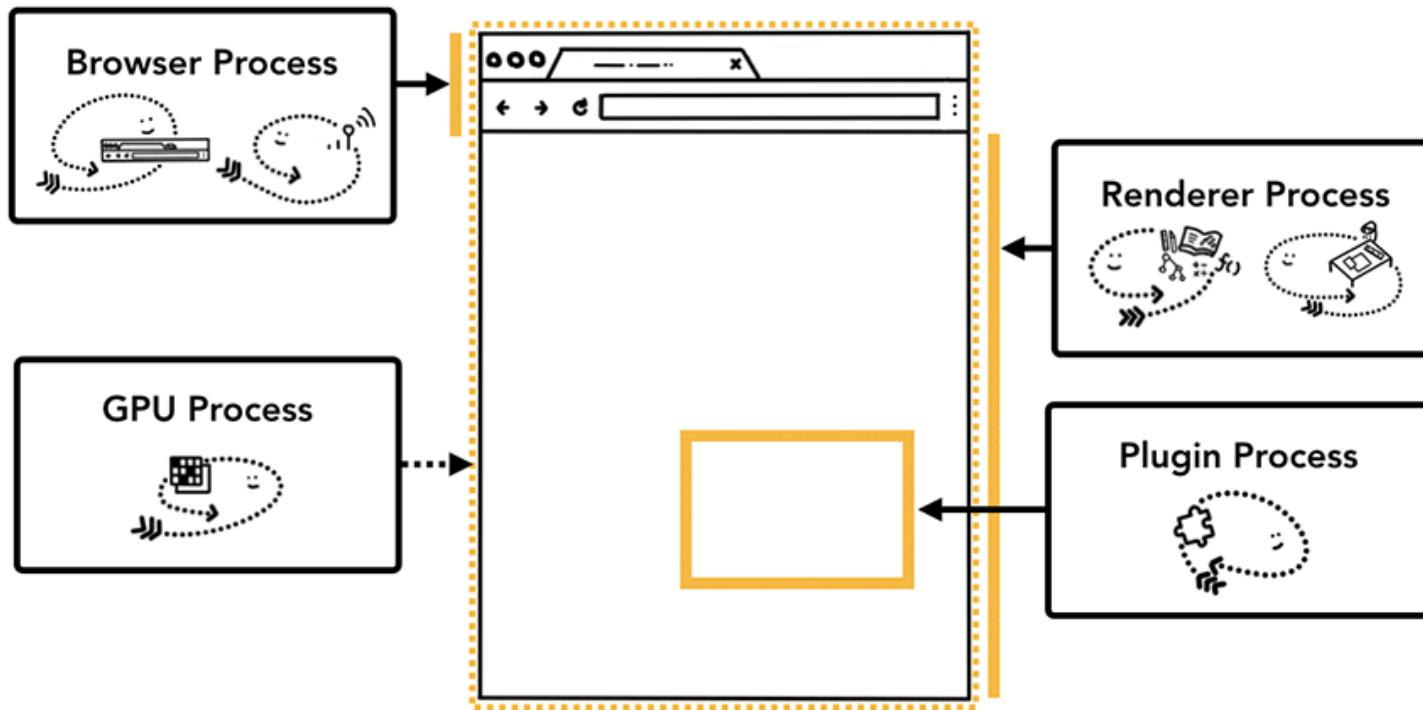
<https://www.chromium.org/developers/design-documents/multi-process-architecture>

Motivation for Chrome

Compromised renderer processes (also known as "arbitrary code execution" attacks in the renderer process) need to be explicitly included in a browser's security threat model. **We assume that determined attackers will be able to find a way** to compromise a renderer process, for several reasons:

- Past experience suggests that potentially **exploitable bugs will be present in future Chrome releases.** There were [10 potentially exploitable bugs in renderer components in M69](#), [5 in M70](#), [13 in M71](#), [13 in M72](#), [15 in M73](#). **This volume of bugs holds steady despite years of investment into developer education, fuzzing, vulnerability reward programs, etc.** Note that this only includes bugs that are reported to us or are found by our team.
- Security bugs can often be made exploitable: even 1-byte buffer overruns [can be turned into an exploit.](#)
- **Deployed mitigations** (like [ASLR](#) or [DEP](#)) are [**not always effective.**](#)

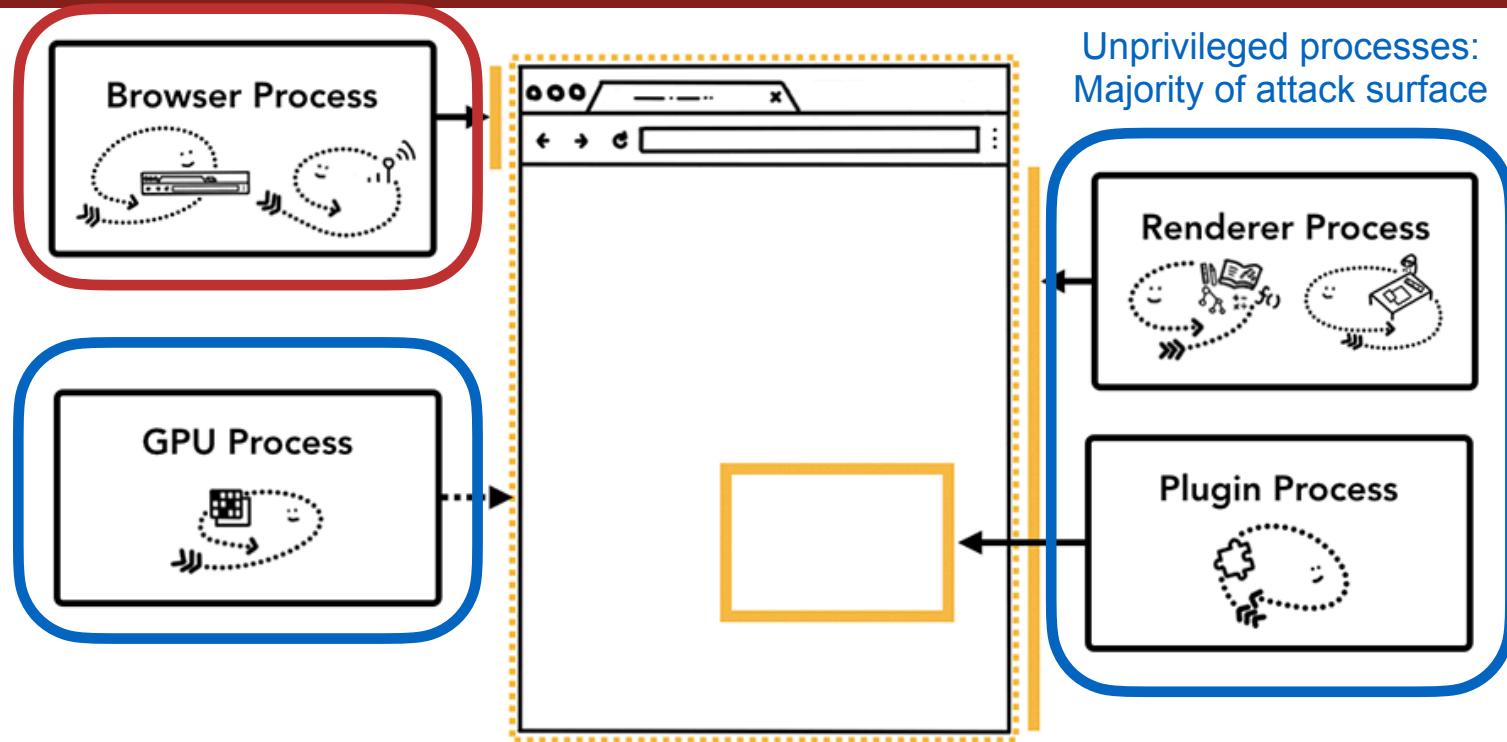
Chrome architecture



REALLY CUTE diagrams from <https://developers.google.com/web/updates/2018/09/inside-browser-part1>
(great read!)

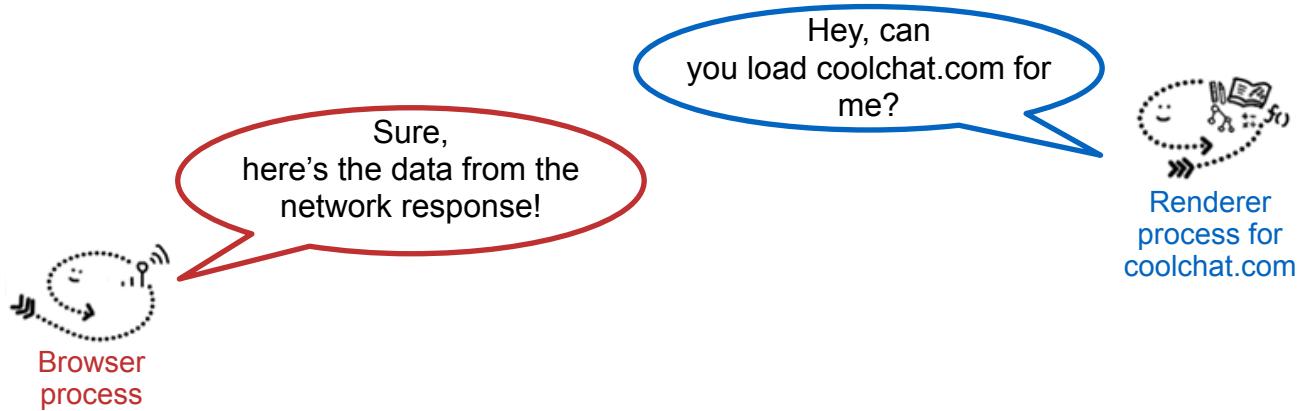
Sandboxing: Defense against RCE

Privileged process:
Minimal attack surface

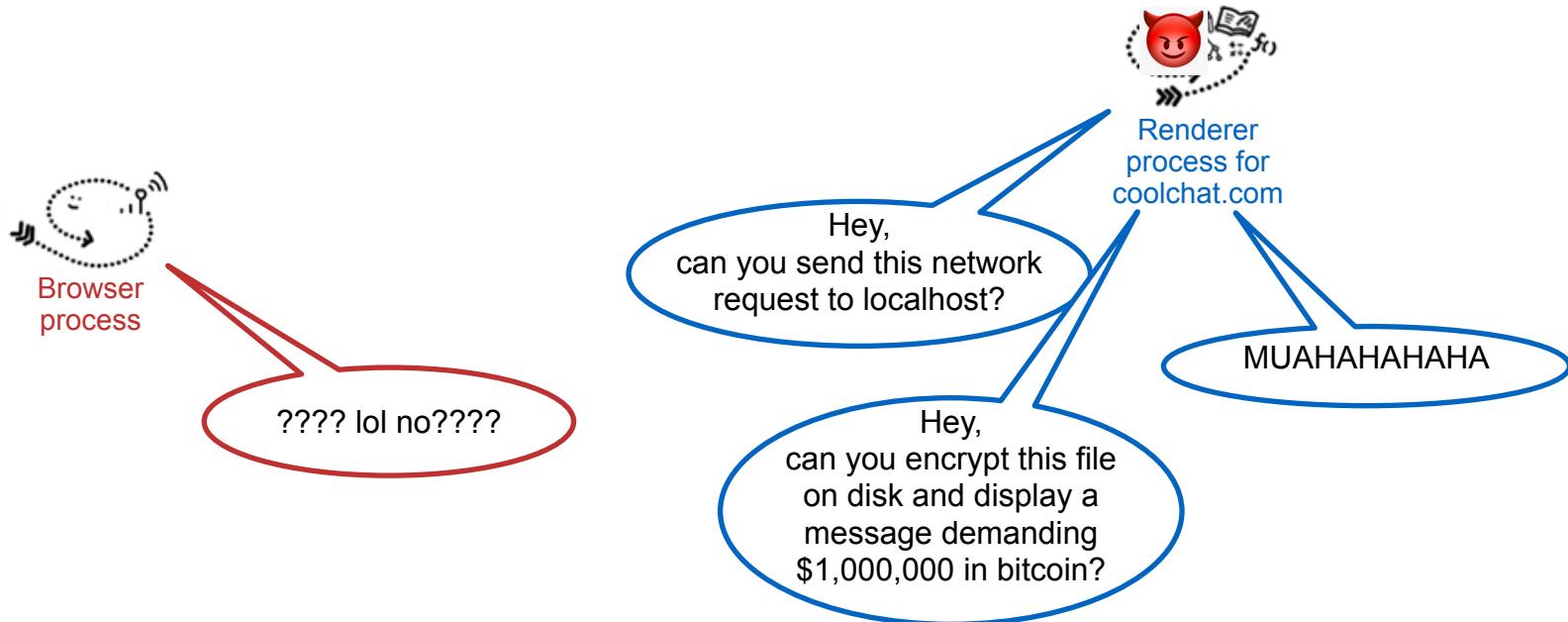


REALLY CUTE diagrams from <https://developers.google.com/web/updates/2018/09/inside-browser-part1>
(great read!)

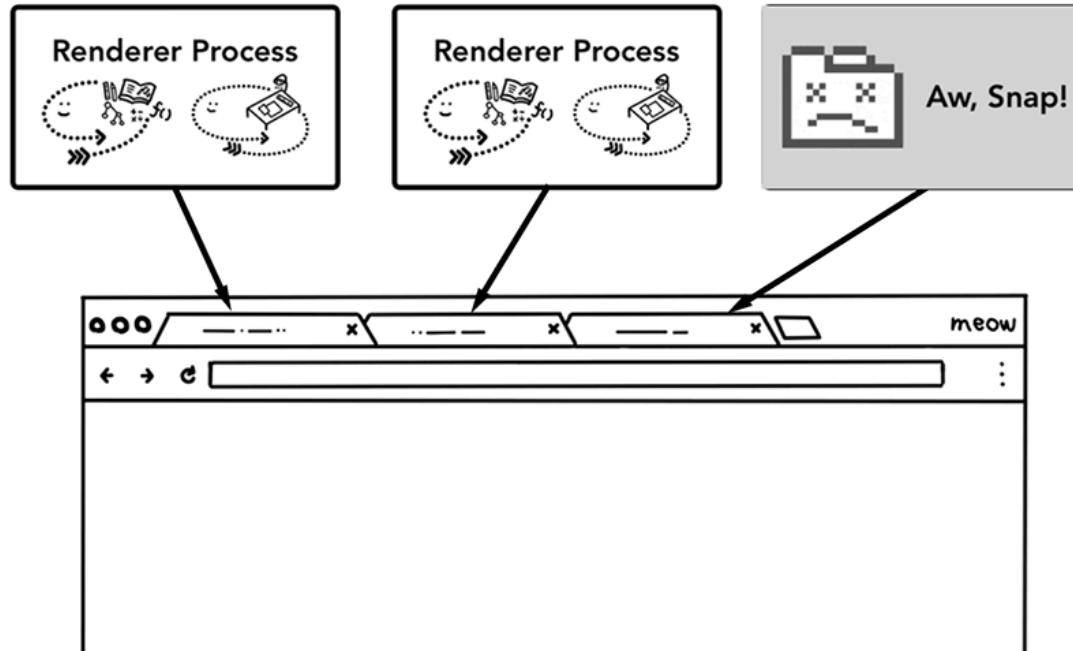
Sandboxing: Defense against RCE



Sandboxing: Defense against RCE

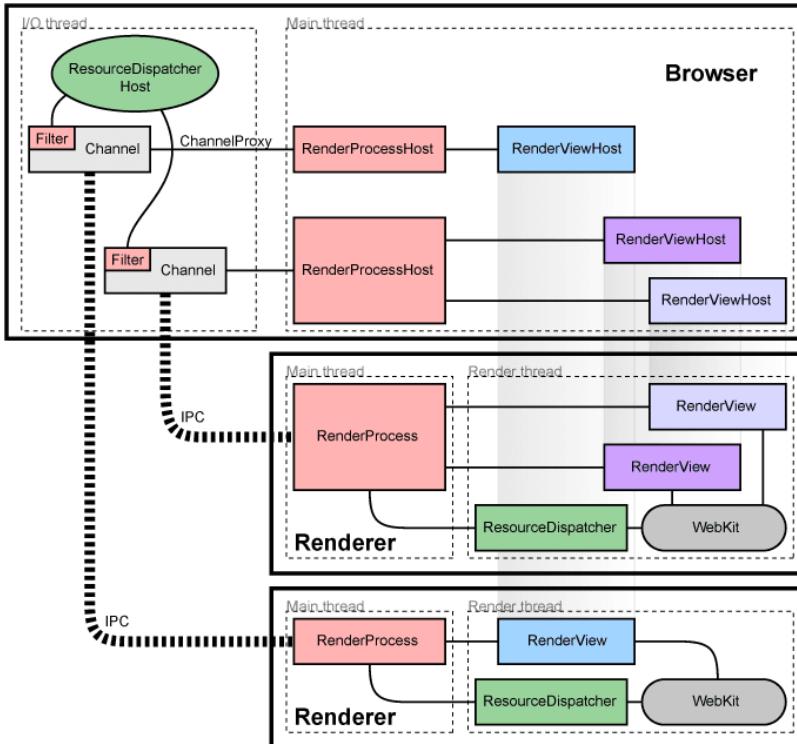


Isolation: Increased robustness



REALLY CUTE diagrams from <https://developers.google.com/web/updates/2018/09/inside-browser-part1>
(great read!)

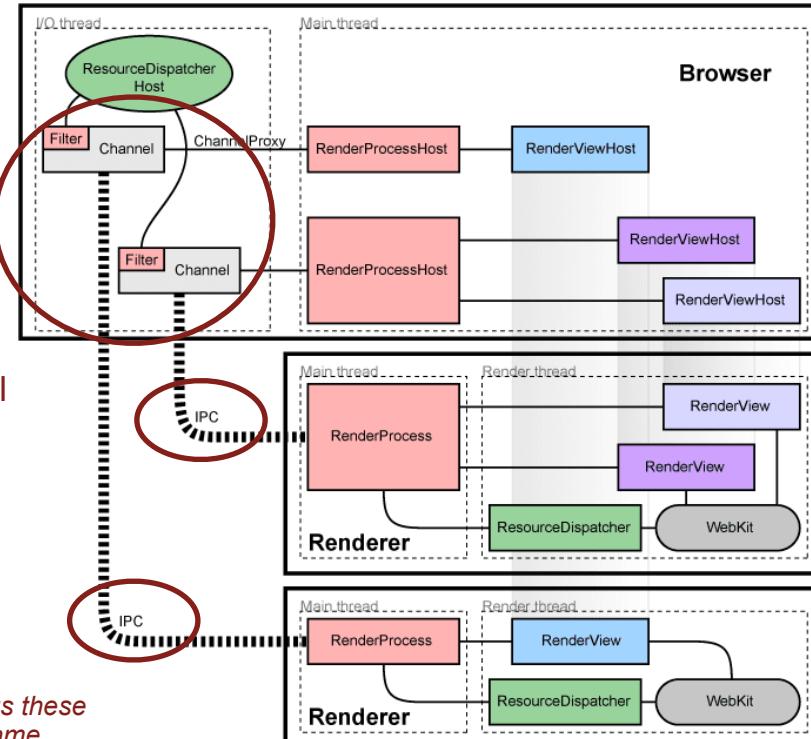
Chrome architecture



Sandboxed processes: no access to network, filesystem, etc

If there is embedded content, may use multiple threads to render that content and manage communication between frames

Chrome architecture



IPC channels = pipes*

Message passing model

Events (e.g. click, keystroke, etc) are relayed through these pipes! No signals

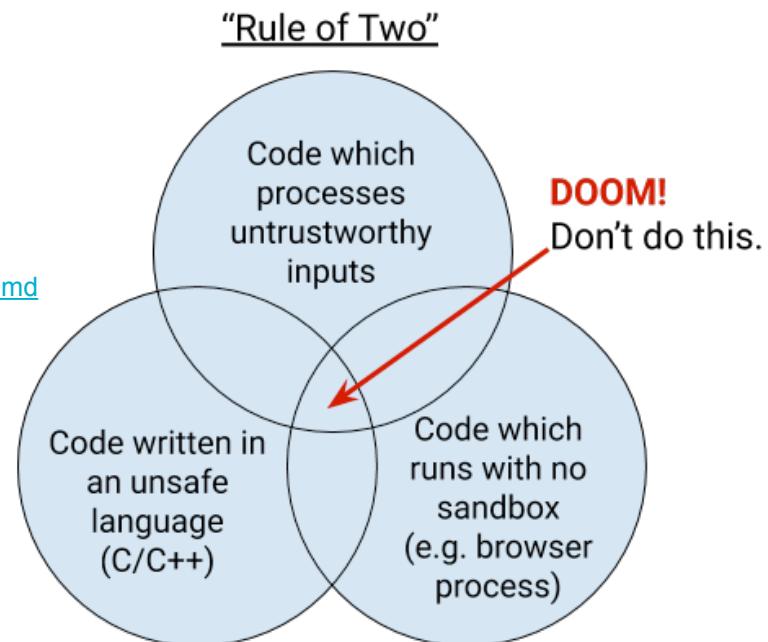
** they use slightly fancier things these days, but the idea is still the same*

Chromium Rule of Two

The Rule Of 2 is: Pick no more than 2 of

- untrustworthy inputs;
- unsafe implementation language; and
- high privilege.

<https://chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md>



This was groundbreaking at the time!

- Really different from how other browsers had been developed

Not good enough

- What does all this work buy us, in terms of isolation?
 - Isolation between tabs
 - Isolation between (potentially malicious) websites and the host
- What does it *not* buy us?
 - Isolation between resources *within* a tab

Embedded content

The screenshot shows a web browser displaying the Stanford Daily website. Several instances of embedded content are highlighted with red boxes:

- Top Left (Red Box):** An Expedia advertisement for travel. It features a yellow background with the Expedia logo, the text "The best trip of your life, all in one app", and a "Book now" button. The Expedia logo is a stylized blue circle with a white 'X' inside.
- Top Right (Red Box):** A photo gallery titled "Photo gallery: Student spirit during Game Week". It shows three thumbnail images of students. The first thumbnail is for "The cultural power of Elizabeth Holmes". The second thumbnail is for "Photo gallery: Student spirit during Game Week". The third thumbnail is for "BEN LEES". Each thumbnail has a "Learn more" button.
- Bottom Right (Red Box):** An advertisement for HBO Max. It features a banner for "THE FALLOUT" with two people lying on the grass. Below the banner, it says "STREAM THE NEW DRAMA NOW" and "HBO MAX SIGN UP NOW". It also mentions "PLANS START AT \$9.99/MONTH".
- Bottom Left (Red Box):** An advertisement for Hertz. It features a banner with the text "Hertz. Let's Go!" and "Save up to 25% off the base rate". Below the banner, there is a "Book Now" button.

The website's sidebar on the left includes sections for "Conference", "Sports", and "Hertz". The main content area shows various news articles and images. The URL "https://stanforddaily.com" is visible at the bottom right of the page.

<https://stanforddaily.com>

Aside: third party resources in general

From CS249i

- Modern websites often rely on “third party resources”
 - Third party: e.g., if you are on stanforddaily.com, any resource served from a domain that is not stanforddaily.com
- A lot of this = ads and analytics
 - Google Analytics appears on ~70% of top websites
 - Also, though — fonts, other utility libraries, images, videos, tweets...
 - Some data [here](#).
- Fun experiment: click “inspect”, click “network”, and load a webpage. What network requests being made to load the webpage? How many of them are for resources served by the domain you’re actually visiting?
 - Ex: loading `latimes` homepage = requests to ~120 domains, for ~330 resources
- Key point: a website isn’t just loading resources from one source!

Embedded content

<http://www.evil.com>

Welcome to Evil!



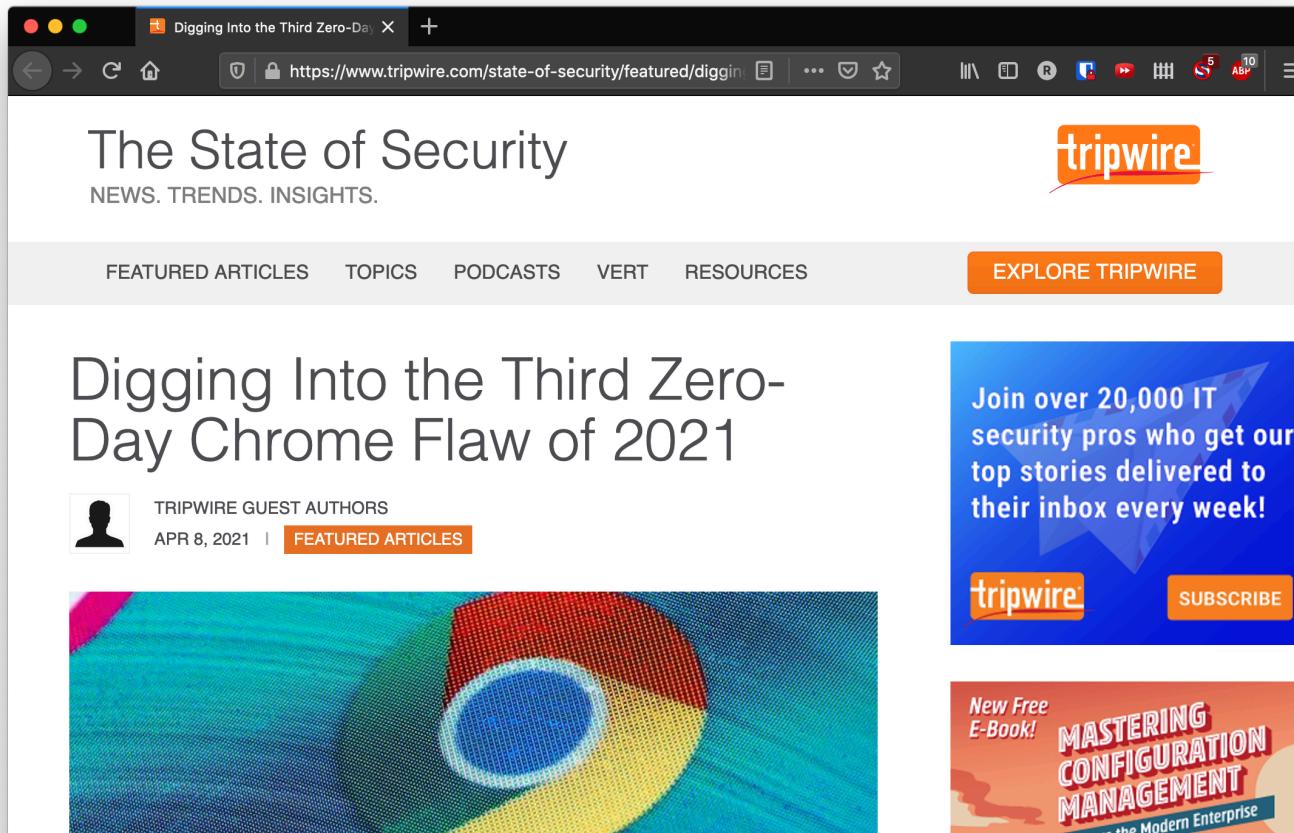
PIN: 1234

Same-origin policy: www.evil.com can embed bank.com, but cannot interact with bank.com or see its data

Embedded content

- Site Isolation Project (2015-2019) aimed to put resources for different origins in different processes
- Extremely difficult undertaking. Cross-frame communication is common (JS postMessage API), and embedded frames need to share render buffers
 - Involved rearchitecting the most core parts of Chrome
- Became especially important in Jan 2018: Spectre and Meltdown
 - When the hardware fails to uphold its guarantees, JS can read arbitrary process memory (even kernel memory, and even if your software has no bugs)!
- Paper/video: <https://www.usenix.org/conference/usenixsecurity19/presentation/reis>

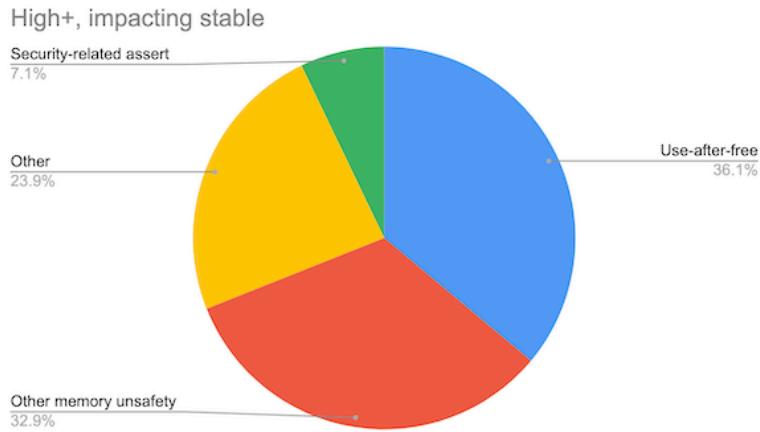
Still not good enough!



The screenshot shows a web browser window with the following details:

- Title Bar:** "Digging Into the Third Zero-Day" (partially visible) and the URL "https://www.tripwire.com/state-of-security/featured/diggin".
- Page Header:** "The State of Security" and "NEWS. TRENDS. INSIGHTS.".
- Page Content:** A large article title "Digging Into the Third Zero-Day Chrome Flaw of 2021". Below it, a "TRIPWIRE GUEST AUTHORS" section with a placeholder profile picture, the date "APR 8, 2021", and a "FEATURED ARTICLES" button. A large image of the Google Chrome logo is displayed.
- Right Sidebar:** A "tripwire" logo, a "EXPLORE TRIPWIRE" button, and a call-to-action box: "Join over 20,000 IT security pros who get our top stories delivered to their inbox every week!" with a "SUBSCRIBE" button.
- Bottom Sidebar:** A "New Free E-Book! MASTERING CONFIGURATION MANAGEMENT" section with a "tripwire" logo and a "SUBSCRIBE" button.

Still not good enough!



- <https://www.chromium.org/Home/chromium-security/memory-safety>
- 70% of high-severity security bugs are caused by memory safety issues

The limits of sandboxing

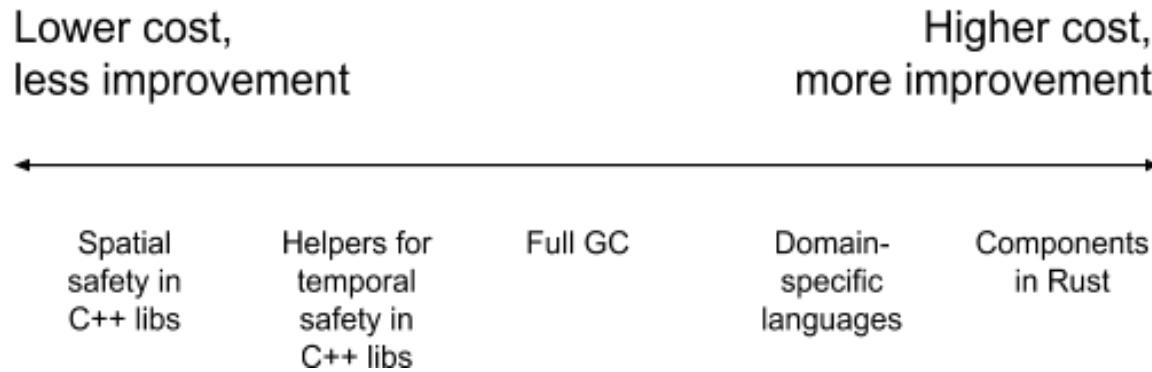
*Chromium's [security architecture](#) has always been designed to assume that these bugs exist, and code is sandboxed to stop them taking over the host machine... But **we are reaching the limits of sandboxing and site isolation.***

A key limitation is that the process is the smallest unit of isolation, but processes are not cheap.

We still have processes sharing information about multiple sites. For example, **the network service is a large component written in C++ whose job is parsing very complex inputs from any maniac on the network. This is what we call “the doom zone” in our [Rule Of 2](#) policy: the network service is a large, soft target and vulnerabilities there are of [Critical](#) severity.**

Just as Site Isolation improved safety by tying renderers to specific sites, we can imagine doing the same with the network service: we could have many network service processes, each tied to a site or (preferably) an origin. That would be beautiful, and would hugely reduce the severity of network service compromise. **However, it would also explode the number of processes Chromium needs, with all the efficiency concerns that raises.**

What we're trying



We expect this strategy will boil down to two major strands:

- Significant changes to the C++ developer experience, with some performance impact. (For instance, **no raw pointers, bounds checks, and garbage collection**.)
- An option of a programming language designed for **compile-time safety checks with less runtime performance impact** – but obviously there is a cost to bridge between C++ and that new language.

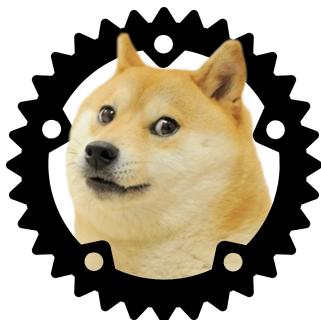
Anatomy of a sandbox escape

- <https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html> (2012 but it's more accessible than some other writeups)
 - First exploit chains together *six bugs* to escape the sandbox
 - Second one uses *ten(!!)*
- <https://googleprojectzero.blogspot.com/2019/04/virtually-unlimited-memory-escaping.html> (2019)

Aside: “Security” vs. “Privacy”

- Even if Google Chrome were “perfectly secure”...
- Can they collect & use data on your browsing history and habits? Yes.
- Can websites save cookies to track your browsing habits and actions, build a detailed profile, target ads to you, etc.? Also yes.
- Do you “know” what you’re getting into when you sign up to use a commercial browser? Hypothetically, yes.
- Is Google going to post your credit card number on the web? Is Google going to leverage your Internet activity to criminalize you? Probably not (bad for business).

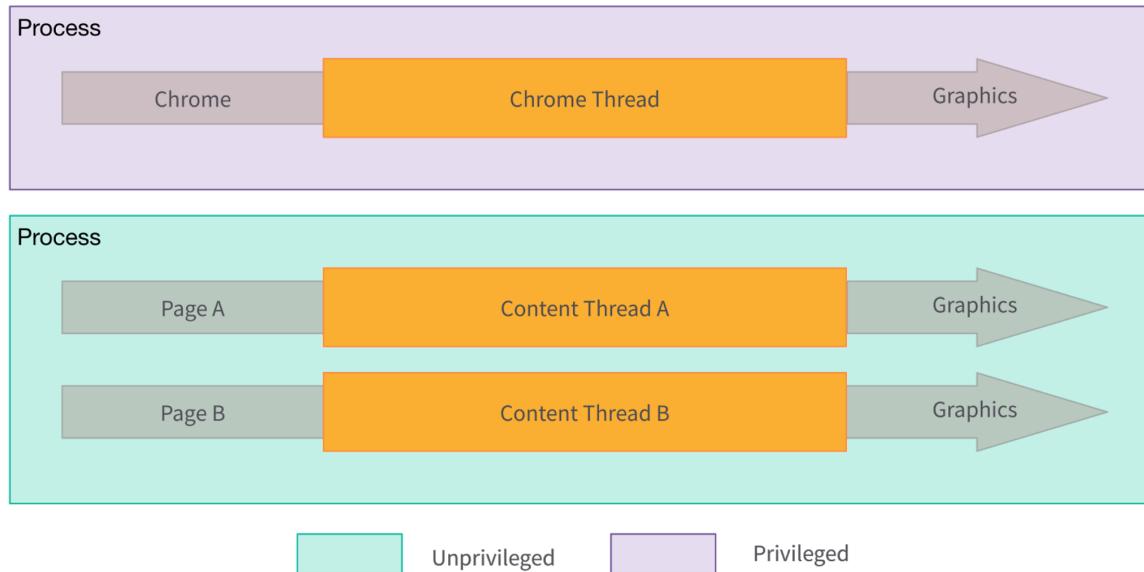
Alternative approach: Servo



Alternative approach

- Wha — this all sounds like a ton of work!
- What if we just implement the browser in a language that helps us avoid these mistakes in the first place?
- Servo is an experimental browser engine from Mozilla Research written in Rust
 - Components of Servo have been gradually adapted in Firefox (Gecko)
 - Note: security was not the primary motivation for Servo, but it's what we're focusing on here

Servo approach



- Have some sandboxing, but don't sweat it too much. Tabs often share processes
- Everything is written in Rust, so we don't have to worry about security issues, right?

Rust does not prevent all bugs

Implications of Rewriting a Browser Component in Rust:

Over the course of its lifetime, there have been 69 security bugs in Firefox's style component. If we'd had a time machine and could have written this component in Rust from the start, 51 (73.9%) of these bugs would not have been possible. While Rust makes it easier to write better code, it's not foolproof.

There are classes of bugs that Rust explicitly does not address—particularly correctness bugs. In fact, during the Quantum CSS rewrite, engineers accidentally reintroduced a critical security bug that had previously been patched in the C++ code, regressing the fix for bug 641731... As a trivial history-stealing bug, this is rated security-high.

Rust code can have memory safety issues too!

- Libraries often use “unsafe Rust” when we need to do things that the compiler can’t guarantee is safe, e.g.:
 - Data structures
 - Implementing new concurrency primitives
 - Running platform-specific assembly instructions
- Testing Firefox with ThreadSanitizer yielded two race conditions in Rust low-level library code: <https://hacks.mozilla.org/2021/04/eliminating-data-races-in-firefox-a-technical-report/>
- Paper on memory and concurrency bugs in unsafe Rust (PDF [here](#)): “Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs” (Qin et. al., 2020).

Limitations of “Rewrite it in Rust!”

- It's impractical to rewrite an entire project in a new language
 - The majority of Firefox is still written in C++
- Rewriting projects introduces bugs (and sometimes reintroduces old, long-fixed bugs)
- Rust code still has security vulnerabilities
 - From correctness issues
 - And even memory safety issues from `unsafe` code

Conclusion

- There is no perfect solution
- We need all the tools we can get:
 - Memory-safe programming languages
 - Sandboxing
 - Fuzzing and dynamic analysis
 - Code review, audits, bug bounty programs?
 - More???

[Bonus slides] How Chrome Does Fork

- <http://neugierig.org/software/chromium/notes/2011/08/zygote.html>
- Fun related bug report: <https://bugs.chromium.org/p/chromium/issues/detail?id=35793>

What steps will reproduce the problem?

1. *Develop a webapp, use chrome's devtools, minding your own business*
2. *In the meantime, let chrome silently autoupdate in the background*

What is the expected result?

Devtools continue working

What happens instead?

Devtools break after refreshing the page after the autoupdate happened.