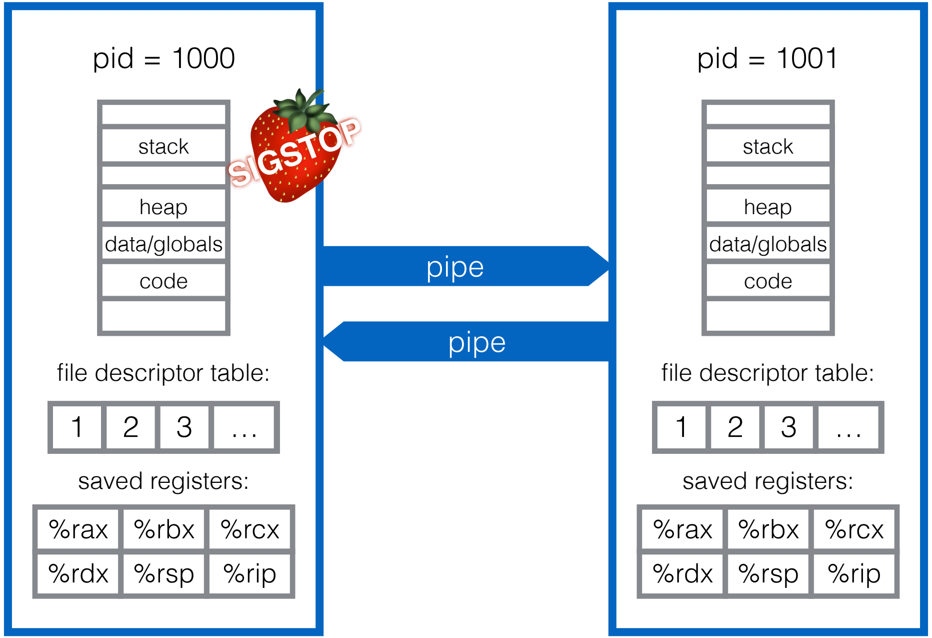
Processes, Threads, and Browser Design

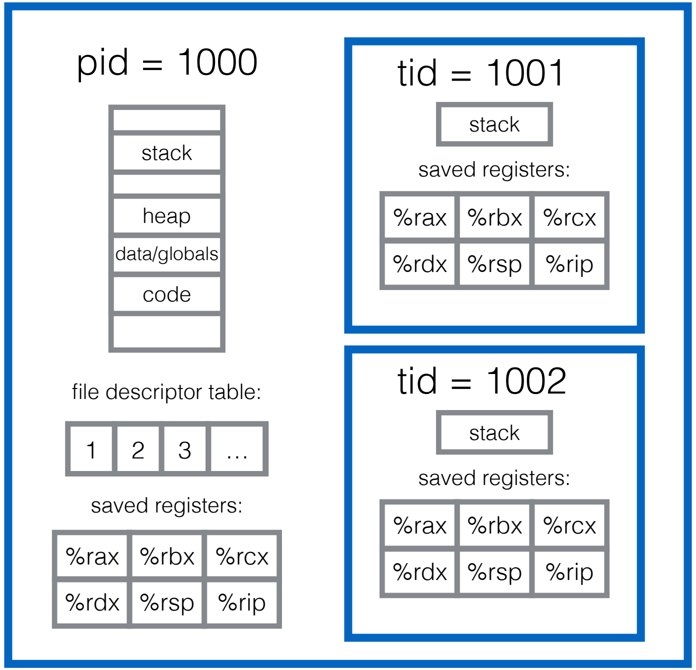
Ryan Eberhardt
July 21, 2021

Processes



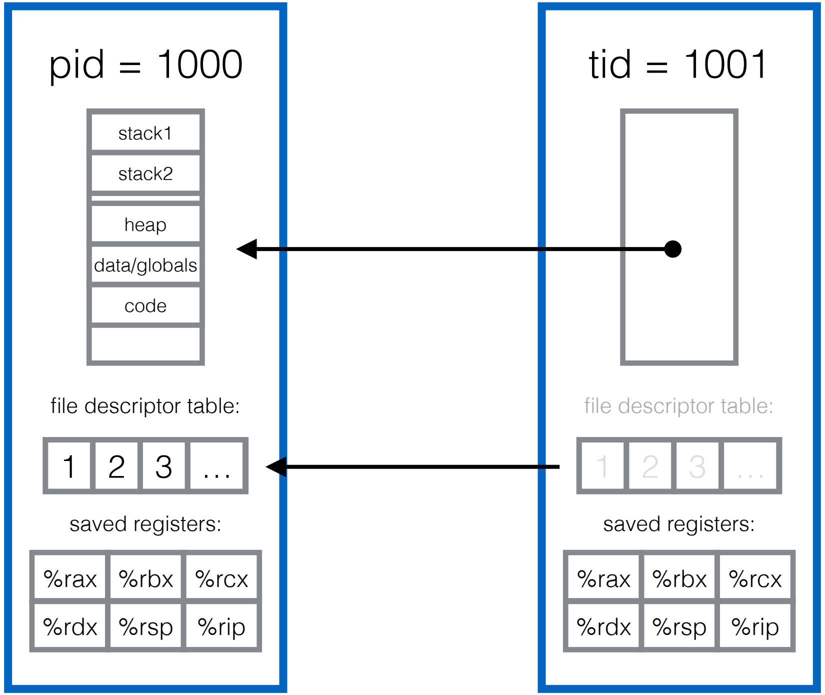
Processes can synchronize using signals and pipes

Threads



Threads are similar to processes; they have a separate stack and saved registers (and a handful of other separated things). But they share most resources across the process

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

Considerations when designing a browser

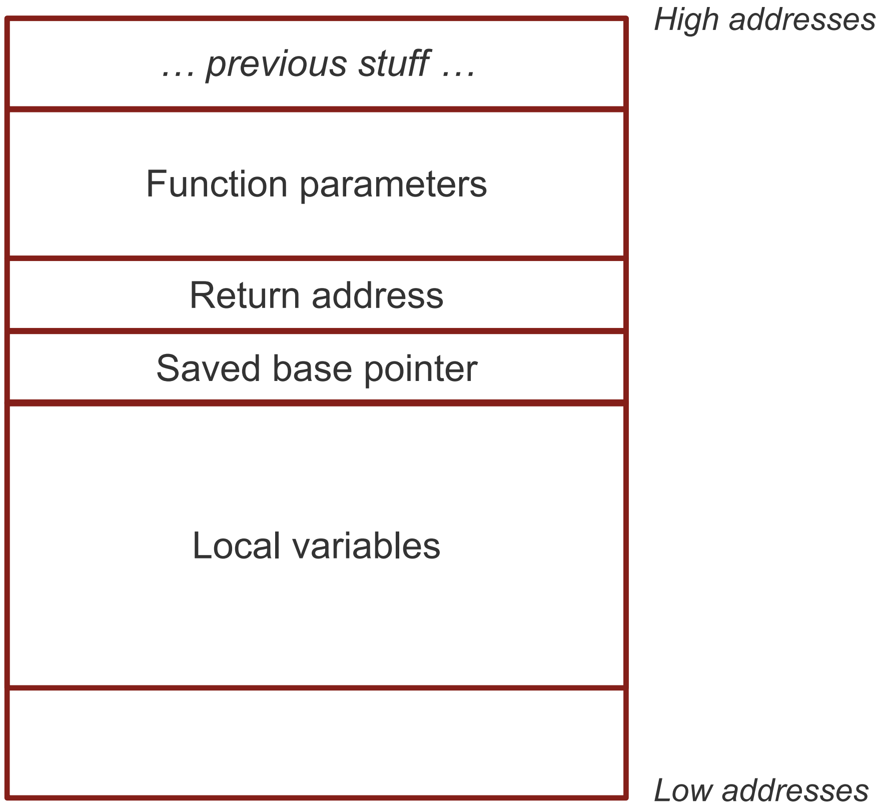
- Speed
- Memory usage
- Battery/CPU usage
- Ease of development
- Security, stability

Considerations when designing a browser

- Speed
  - Typically faster to share memory and to use lightweight synchronization primitives
  - Processes incur additional context switching overhead
- Memory usage
  - Processes use more memory
- Battery/CPU usage
  - 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.

How simple buffer overflows work

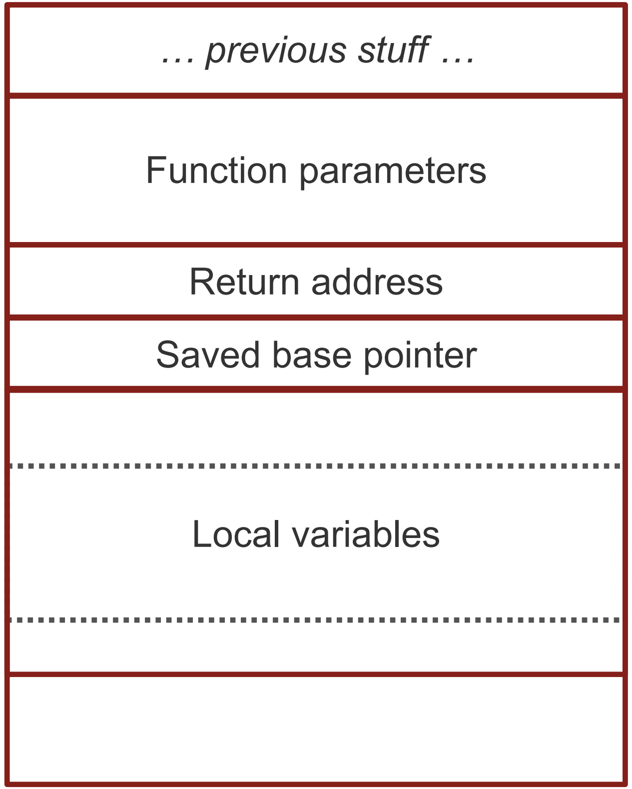
```
; push call arguments, in reverse
push    3
push    2
push    1
call    callee    ; call subroutine 'callee'

    callee:
    push    ebp         ; save old call frame
    mov     ebp, esp    ; initialize new call frame
    ...do stuff...
    mov     esp, ebp
    pop     ebp         ; restore old call frame
    ret                 ; return


add     esp, 12  ; remove call arguments from frame
```



From https://en.wikipedia.org/wiki/X86_calling_conventions#cdecl

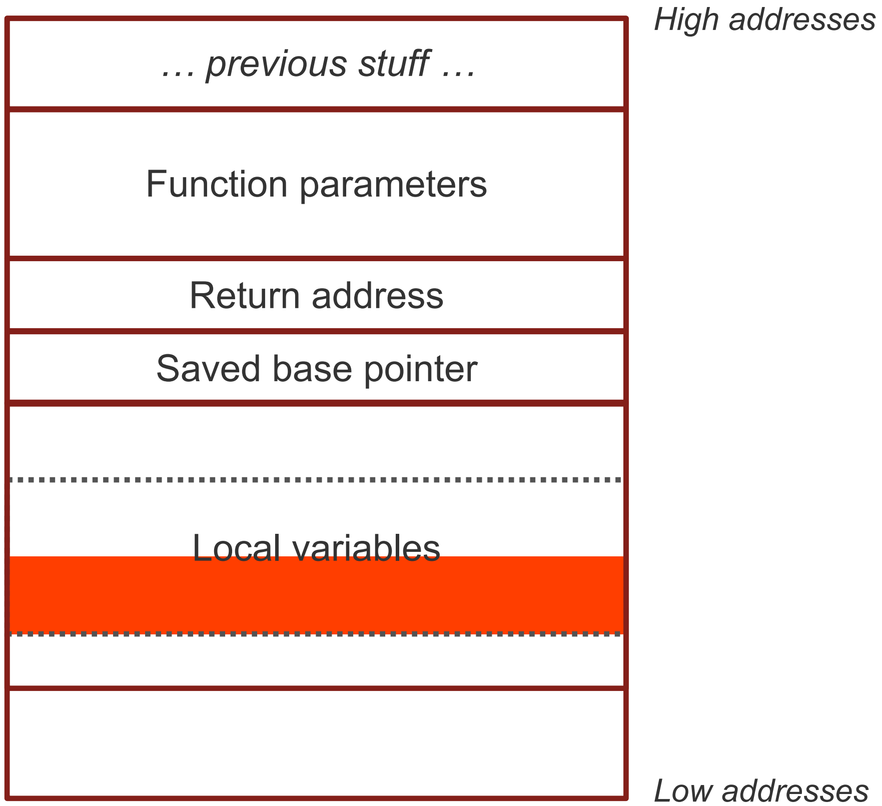
How simple buffer overflows work

```
; push call arguments, in reverse
push    3
push    2
push    1
call    callee     ; call subroutine ‘callee'

    callee:
    push    ebp        ; save old call frame
    mov     ebp, esp   ; initialize new call frame
    ...do stuff...
```



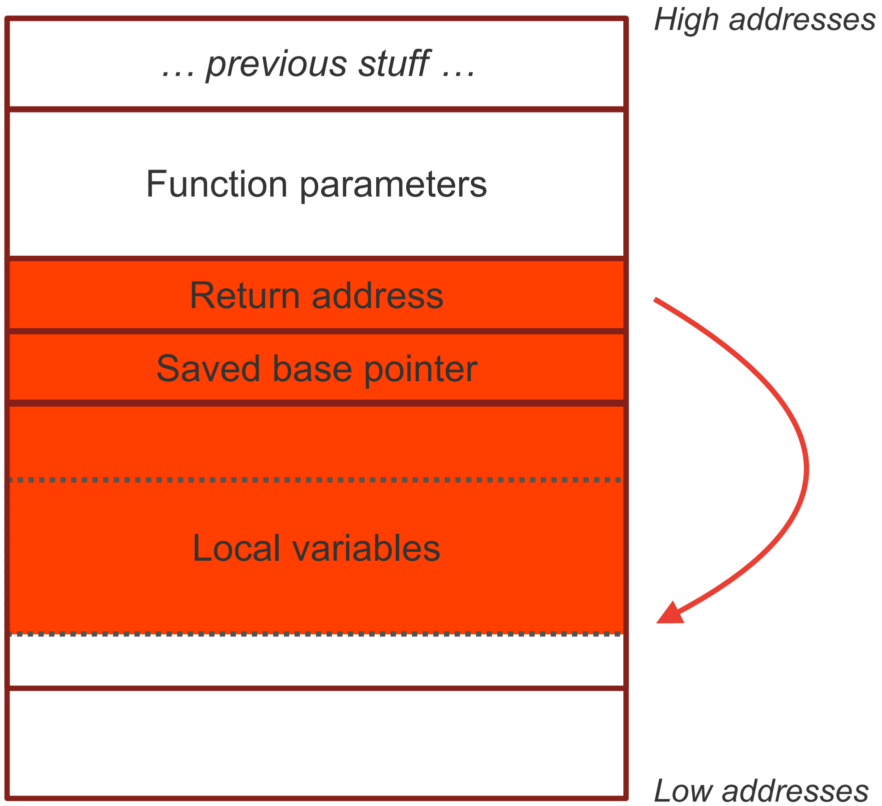
How simple buffer overflows work

```
; push call arguments, in reverse
push    3
push    2
push    1
call    callee    ; call subroutine ‘callee'

    callee:
    push    ebp       ; save old call frame
    mov     ebp, esp  ; initialize new call frame
    ...do stuff...
```



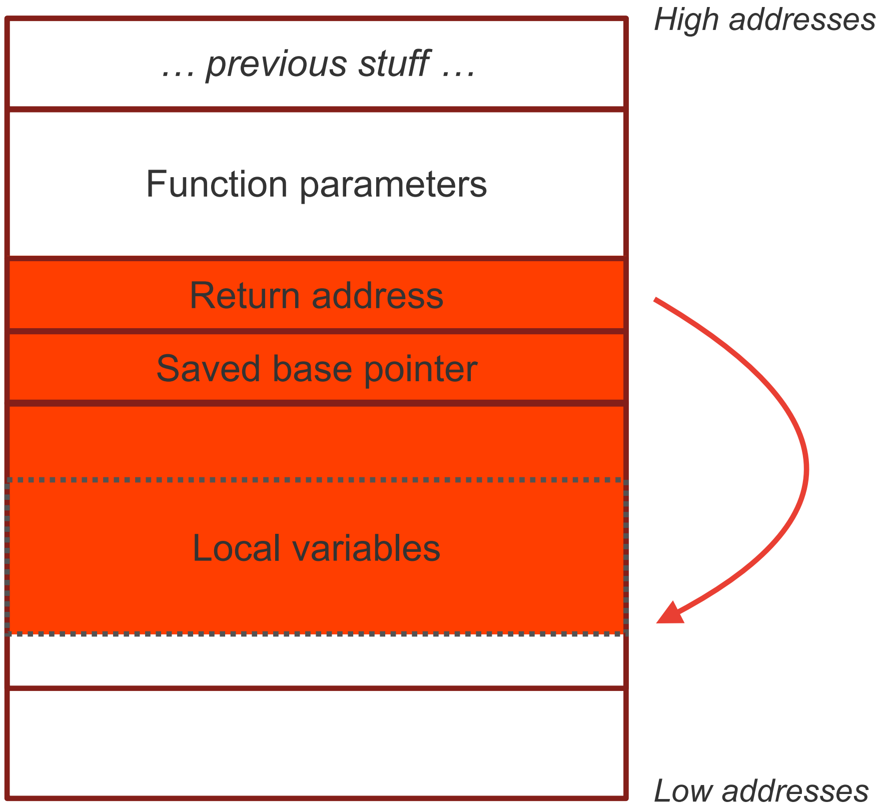
How simple buffer overflows work

```
; push call arguments, in reverse
push    3
push    2
push    1
call    callee    ; call subroutine 'callee'

    callee:
    push    ebp       ; save old call frame
    mov     ebp, esp  ; initialize new call frame
    ...do stuff...
```



How simple buffer overflows work

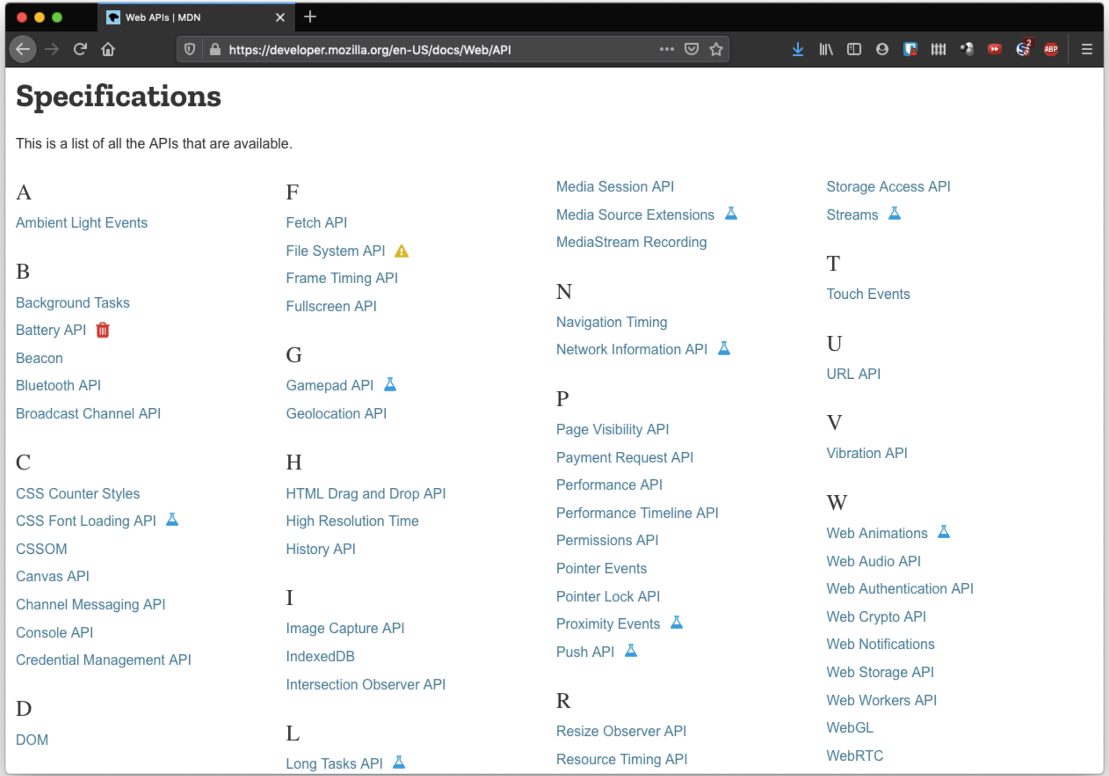
```
; push call arguments, in reverse
push    3
push    2
push    1
call    callee      ; call subroutine 'callee'

    callee:
    push    ebp         ; save old call frame
    mov     ebp, esp    ; initialize new call frame
    ...do stuff...
    mov     esp, ebp
    pop     ebp         ; restore old call frame
    ret                 ; return
```



Any memory corruption can lead to RCE

- This kind of buffer overflow (stack-based buffer overflow overwriting the return address) is the easiest to understand, but most buffer overflows these days are way more subtle
- Even a one-byte overflow can be used to get remote code execution: https://googleprojectzero.blogspot.com/2016/12/chrome-os-exploit-one-byte-overflow-and.html
- If you have *any* memory corruption, you should assume that an attacker with enough determination will be able to figure out how to use it to get RCE

Modern browsers are essentially operating systems



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

Modern browsers are essentially operating systems

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

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.*

https://www.chromium.org/Home/chromium-security/site-isolation

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.*

https://www.chromium.org/Home/chromium-security/site-isolation

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.*

https://www.chromium.org/Home/chromium-security/site-isolation

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.*

https://www.chromium.org/Home/chromium-security/site-isolation

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.*

https://www.chromium.org/Home/chromium-security/site-isolation

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.*

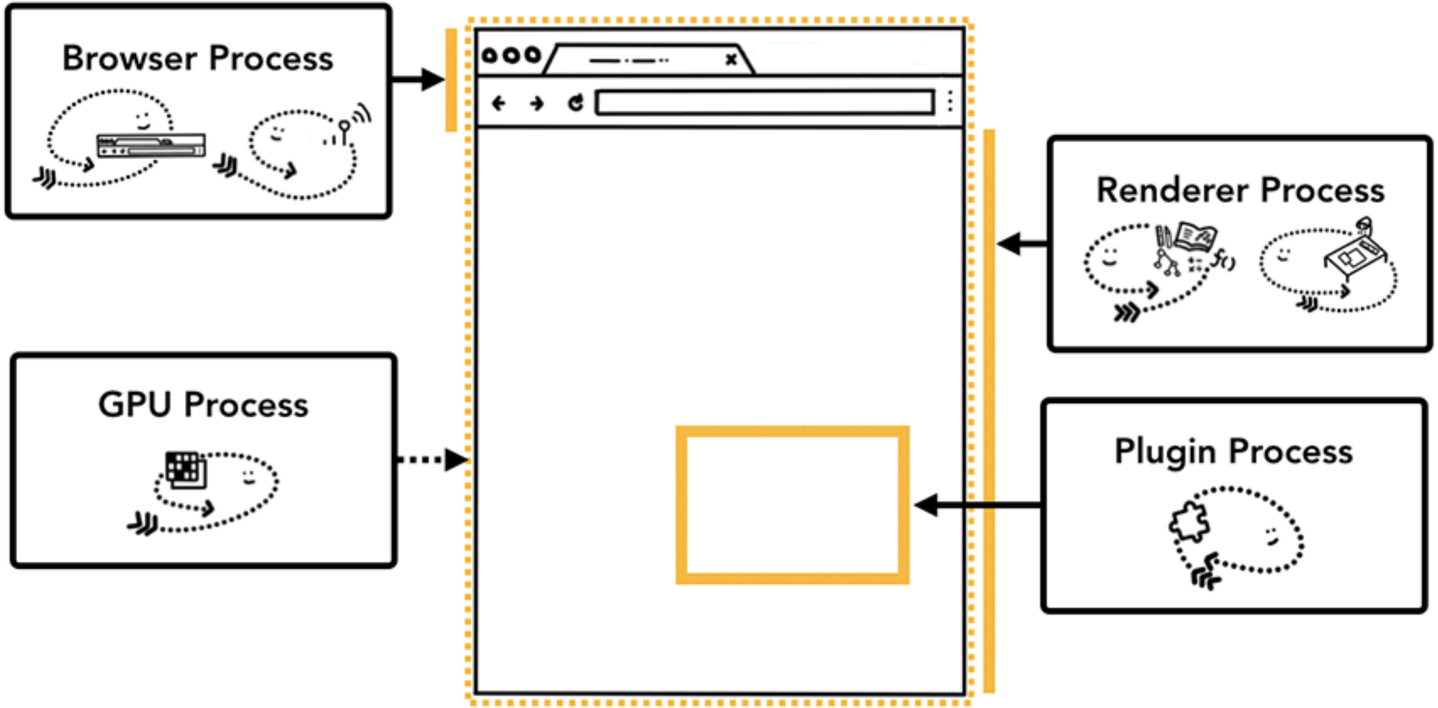
https://www.chromium.org/Home/chromium-security/site-isolation

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.***

https://www.chromium.org/Home/chromium-security/site-isolation

Chrome architecture

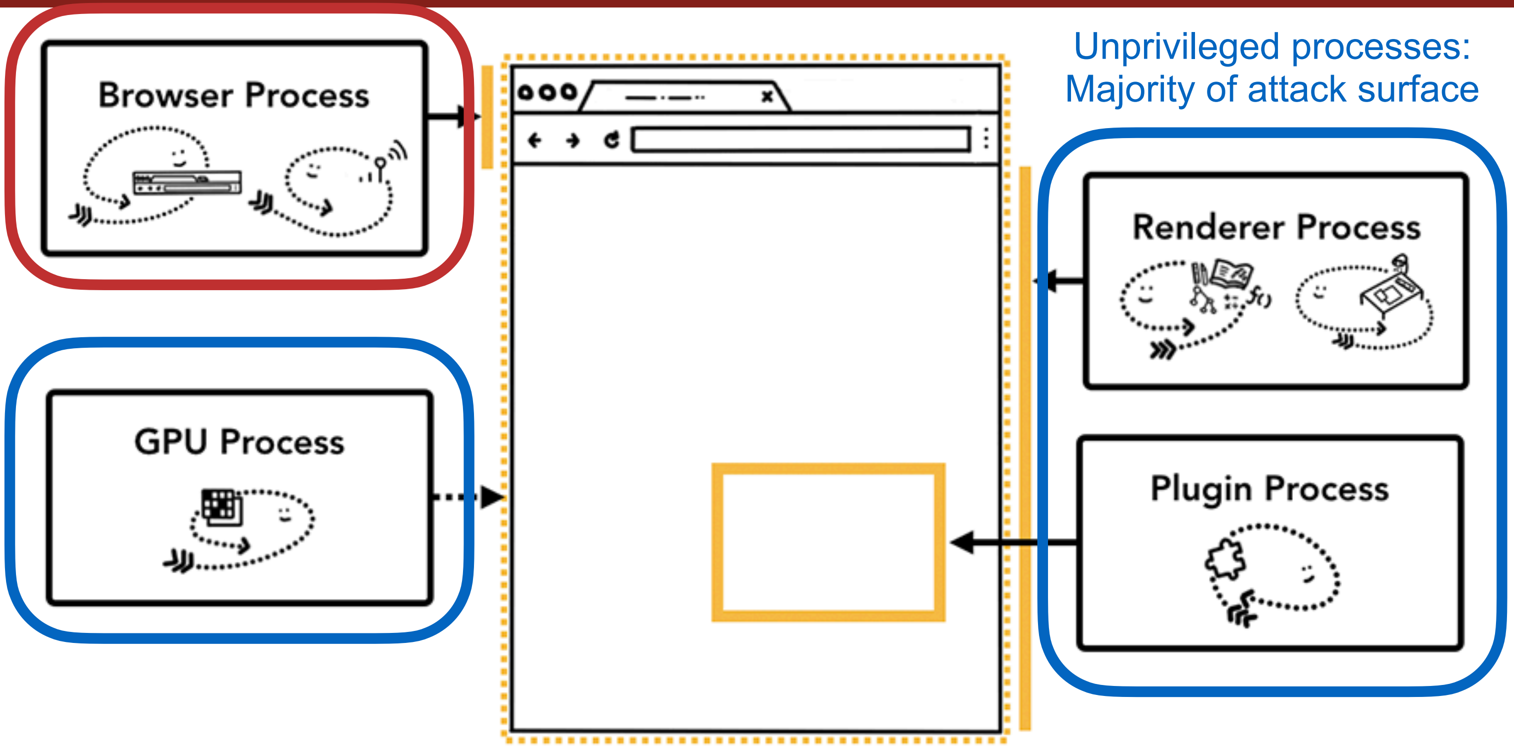


REALLY CUTE diagrams from https://developers.google.com/web/updates/2018/09/inside-browser-part1
(great read!)

Sandboxing: Defense against RCE

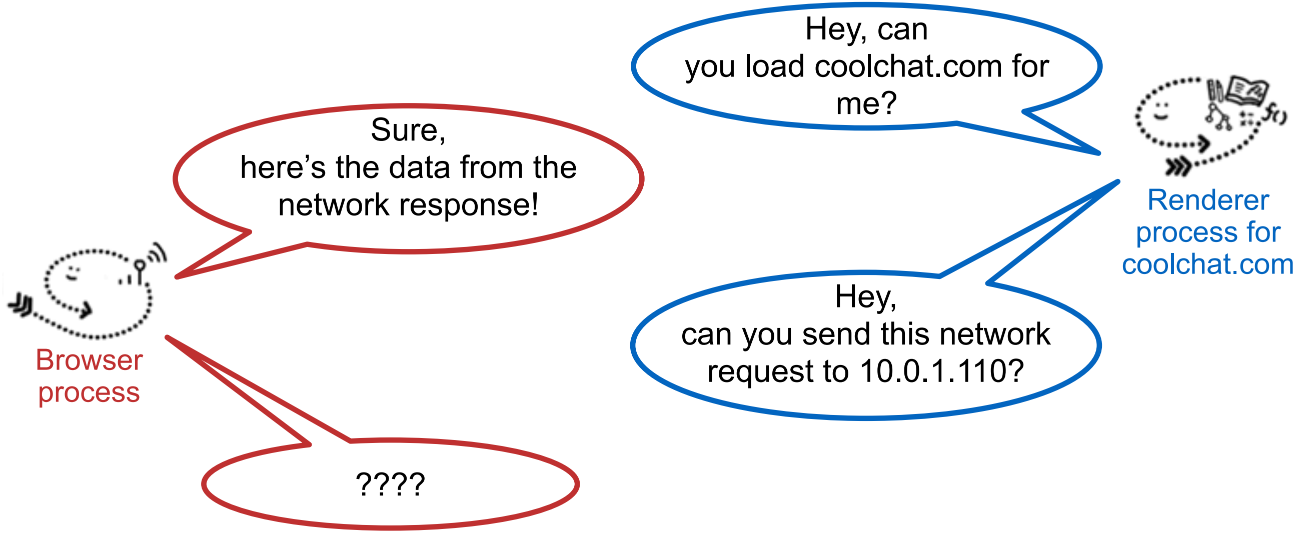
Privileged process: Minimal attack surface

Unprivileged processes: Majority of attack surface

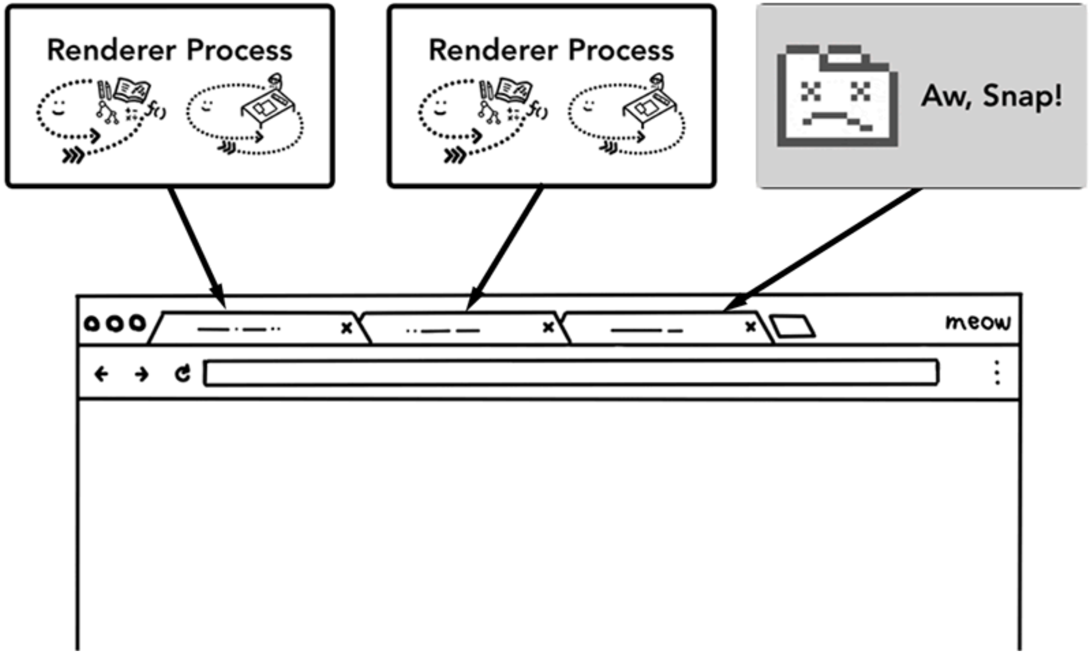


REALLY CUTE diagrams from https://developers.google.com/web/updates/2018/09/inside-browser-part1 (great read!)

Sandboxing: Defense against RCE

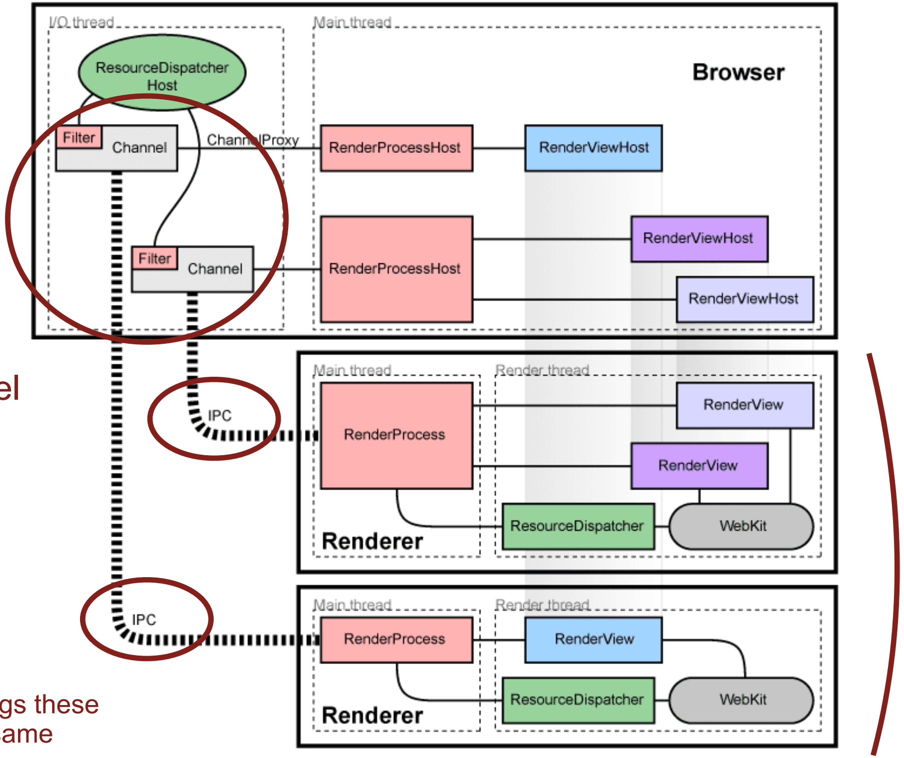


Isolation: Increased robustness



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

Chrome architecture



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

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

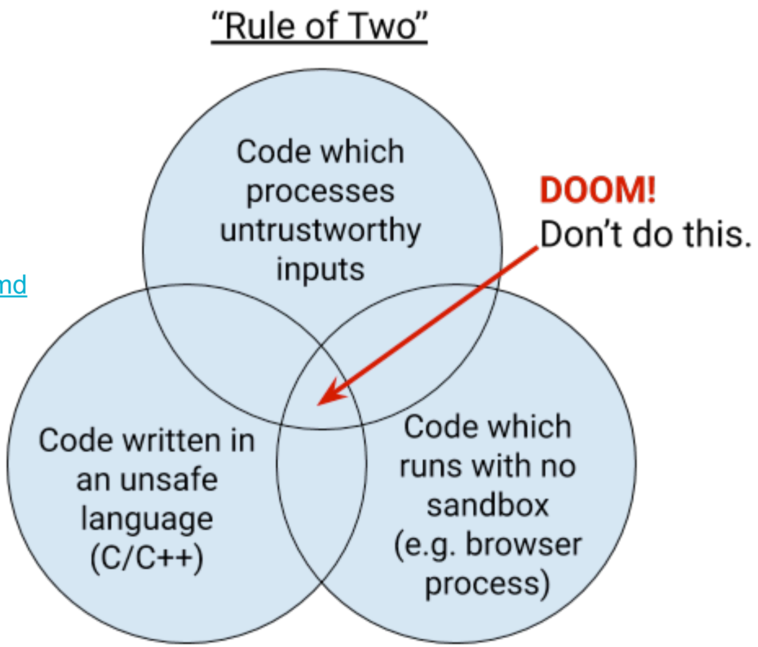
https://www.chromium.org/developers/design-documents/multi-process-architecture (slightly out of date)

Chromium Rule of Two

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

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

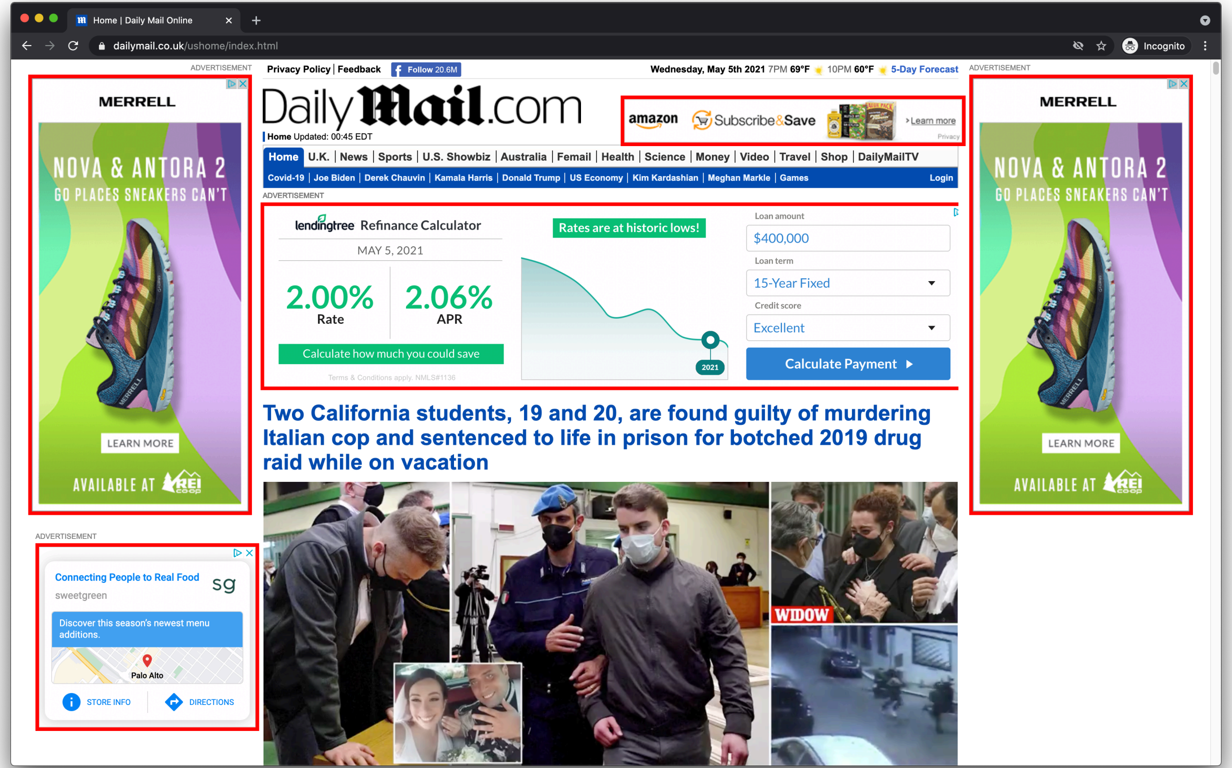
https://chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md



Not good enough

- What does all this work buy us?
  - 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



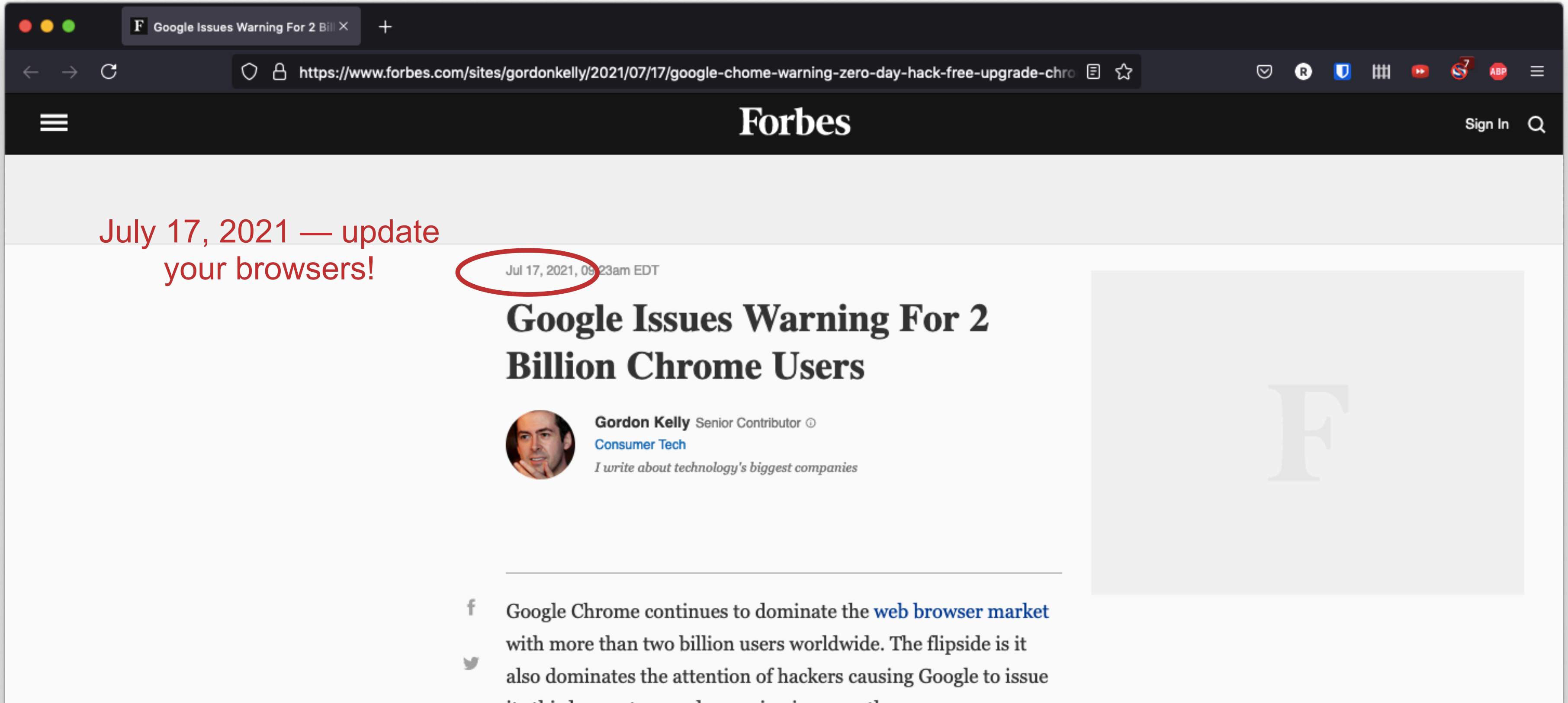
Embedded content

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

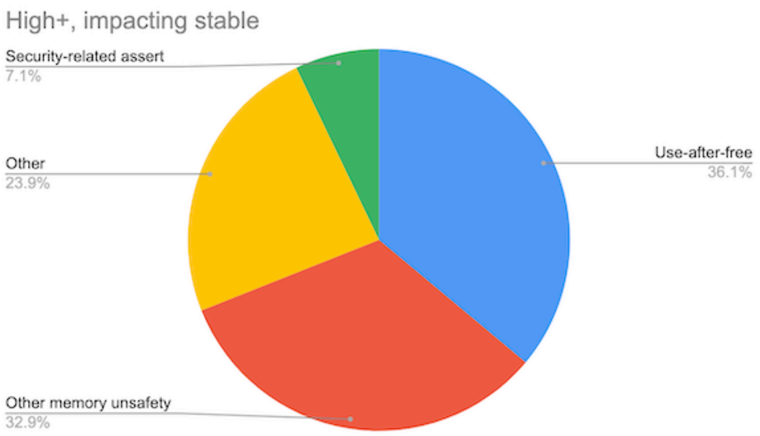
Embedded content

- 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!



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

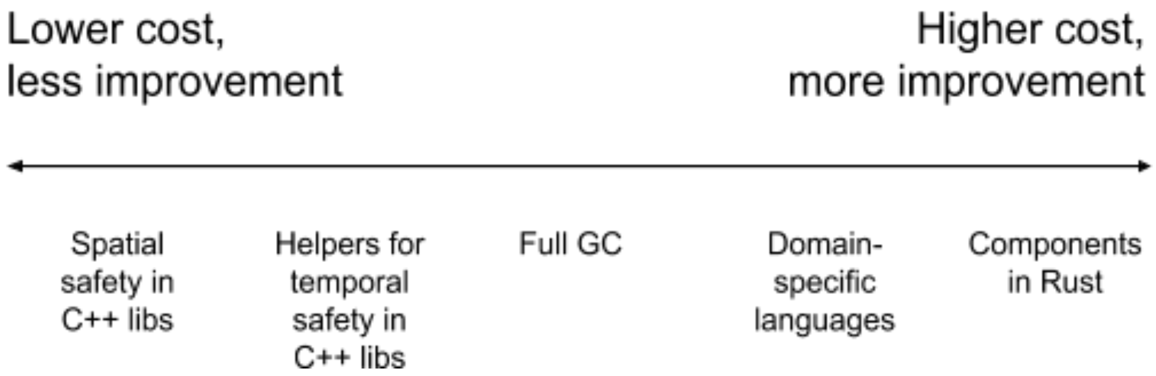
*Chromium's security architecture has always been designed to assume that these bugs exist, and code is sandboxed to stop them taking over the host machine… But **we are reaching the limits of sandboxing and site isolation.***

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)