ENIGMA

The Limits Of Sandboxing

And Next Steps

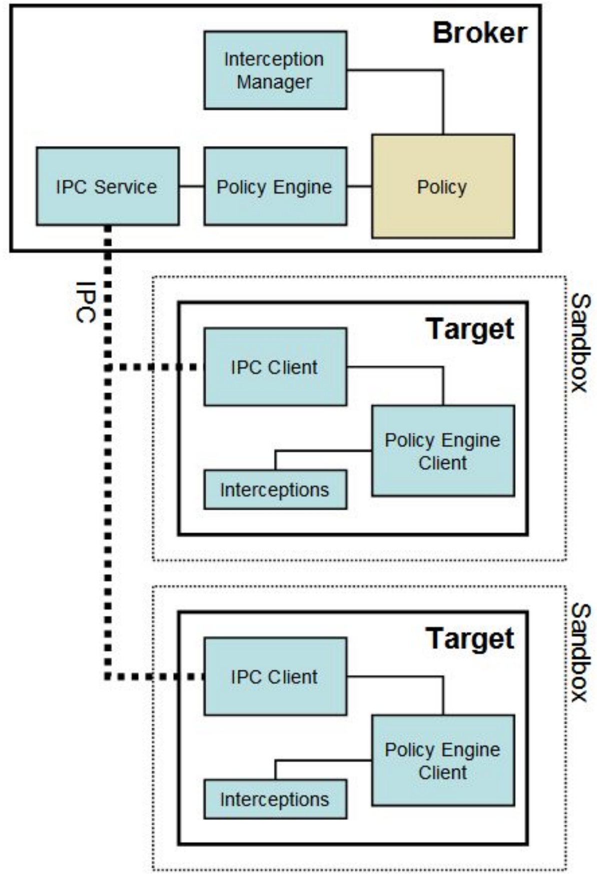
Chris Palmer, Google Chrome Security

Link to slides with speaker notes

There are many links in the notes!

About

What Is Sandboxing


Broker
Interception Manager
IPC Service
Policy Engine
Policy
IPC
Sandbox
Target
IPC Client
Policy Engine Client
Interceptions
Sandbox
Target
IPC Client
Policy Engine Client
Interceptions

A simplified view

Good Sandboxing Is Table Stakes

Extremely necessary to contain a variety of classes of bugs, leaks, and vulnerabilities.

It is not, by itself, sufficient for Chromium, nor arbitrarily applicable.

We’re pretty sure Chromium is near the practical limit…

...but your application might still have significant sandboxing headroom left. Use it!

How To Build A Sandbox

The fundamental building block available to us is the process boundary.

- Fault isolation
- System call filtering
- Low-privilege principals (UID/GID, SID, Access Tokens)

Segmented memory someday...?

OS Mechanisms (Android)

- `isolatedProcess`es
- Medium-grained system call filtering (certain predefined Seccomp-BPF policies)
- SELinux

OS Mechanisms (Linux, Chrome OS)

- Fine-grained system call filtering (freer use of Seccomp-BPF)
- User/PID/network namespaces
- Legacy setuid helper where namespaces not available

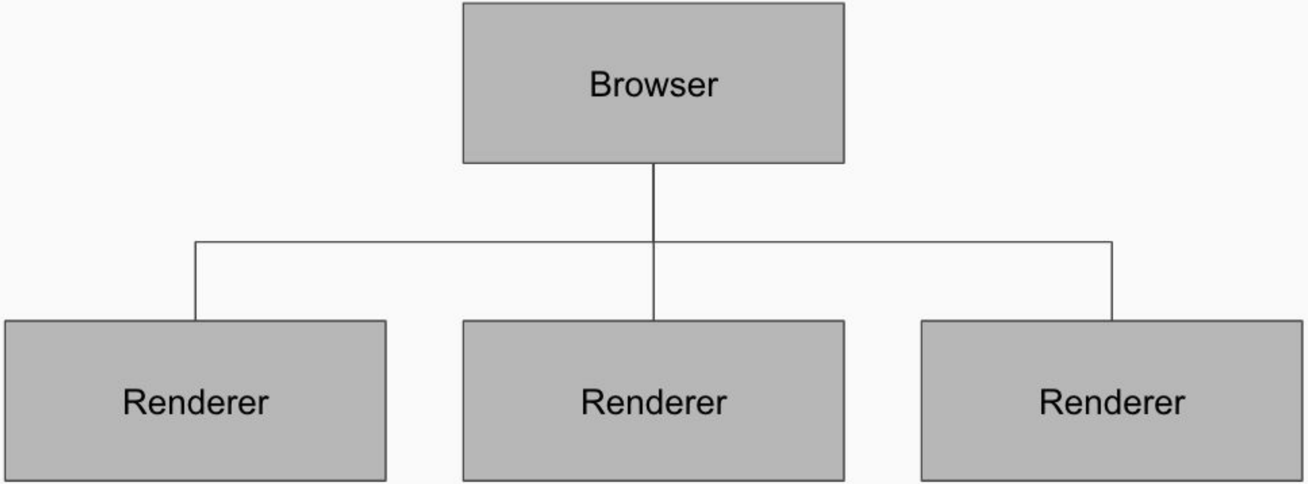
Limitations And Costs

Sadly, you can’t necessarily sandbox everything, nor at a sufficiently fine grain.

- Process space overhead
    - Large on Windows
    - Very large on Android
- Process startup latency
    - High on Windows
    - Very high on Android

Site Isolation

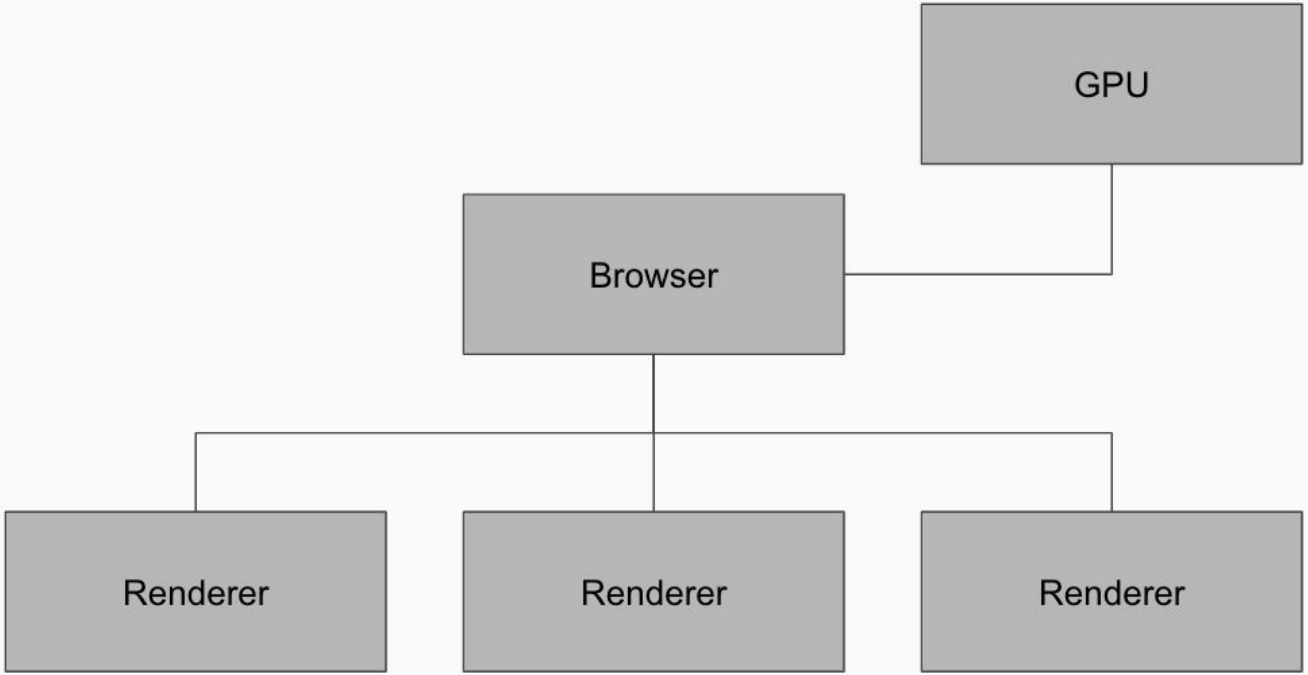
How do we decide when to create a new renderer process?


Browser

Renderer

Renderer

Renderer

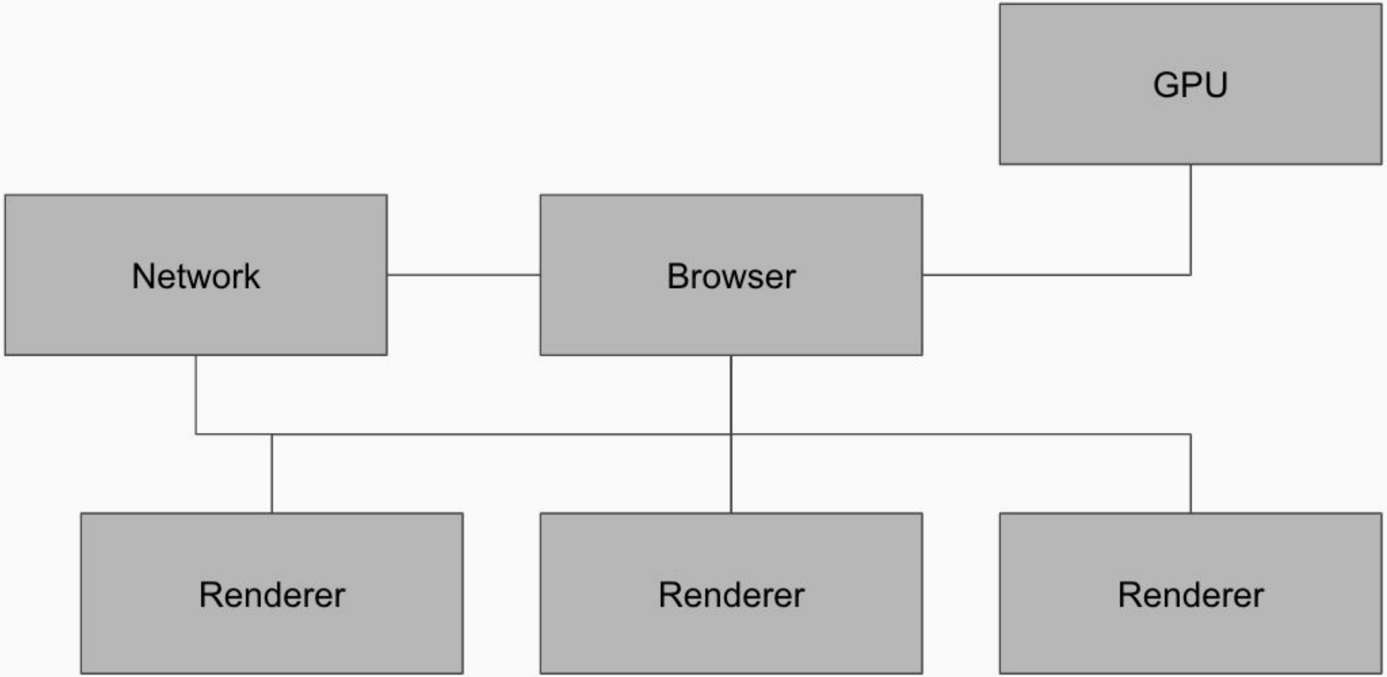

GPU

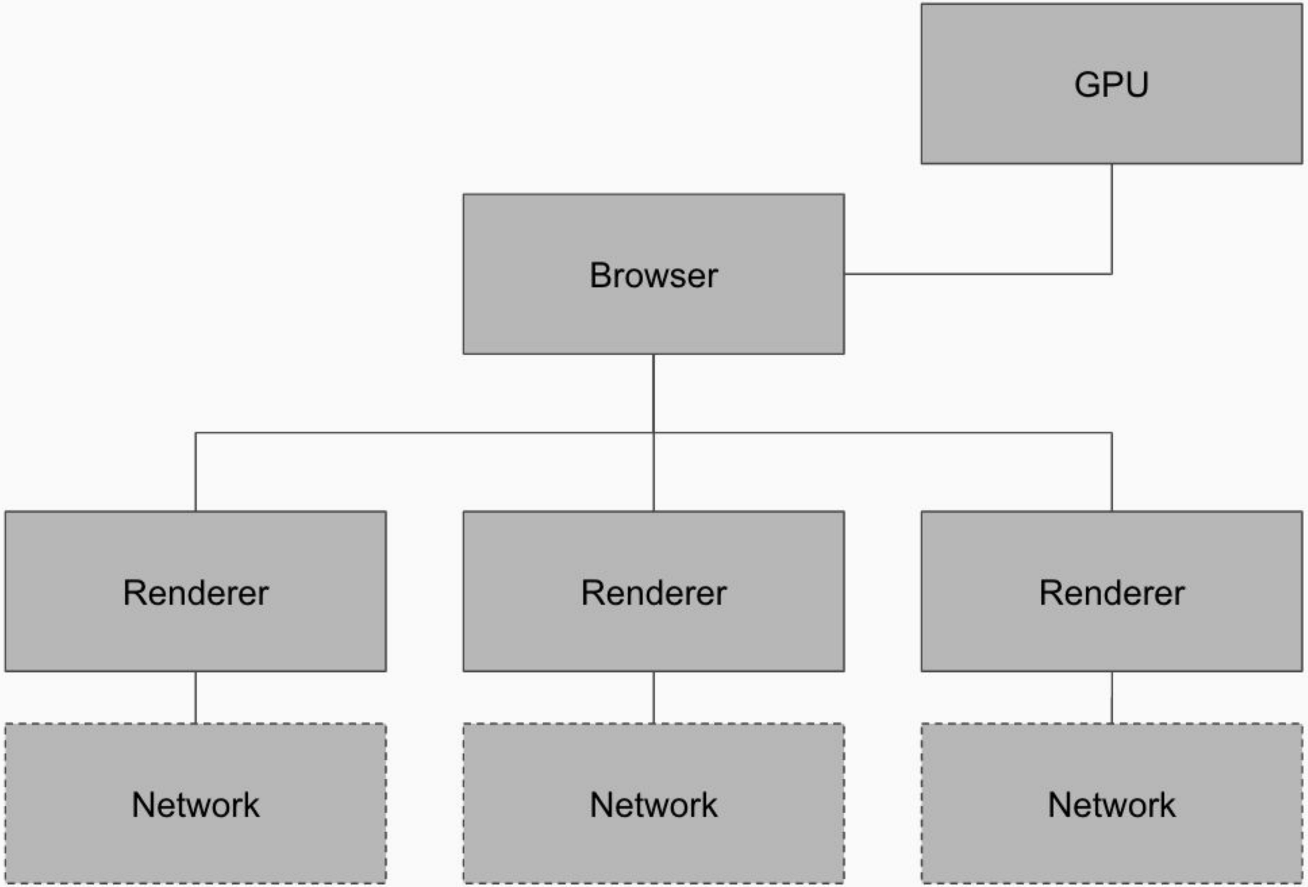
Browser

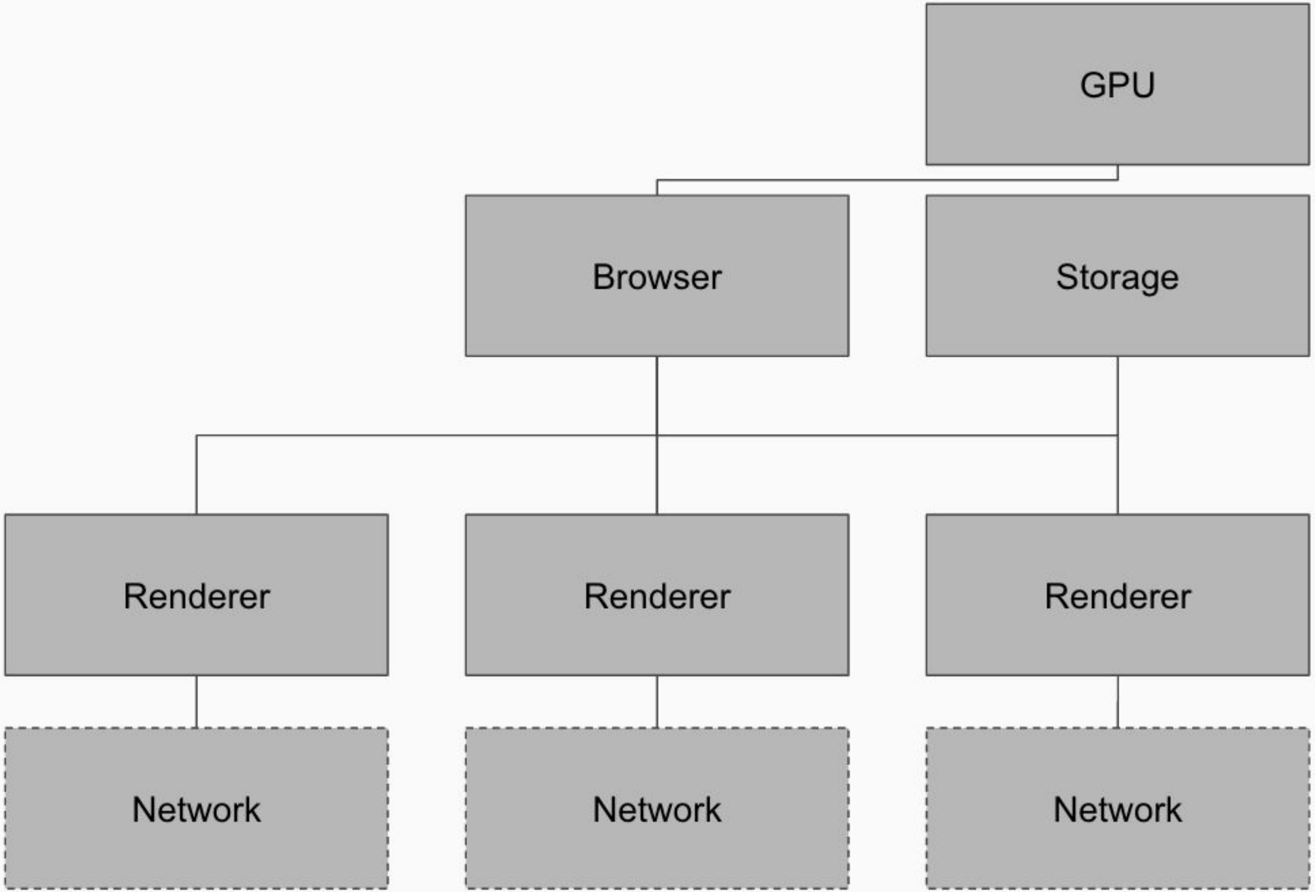
Renderer

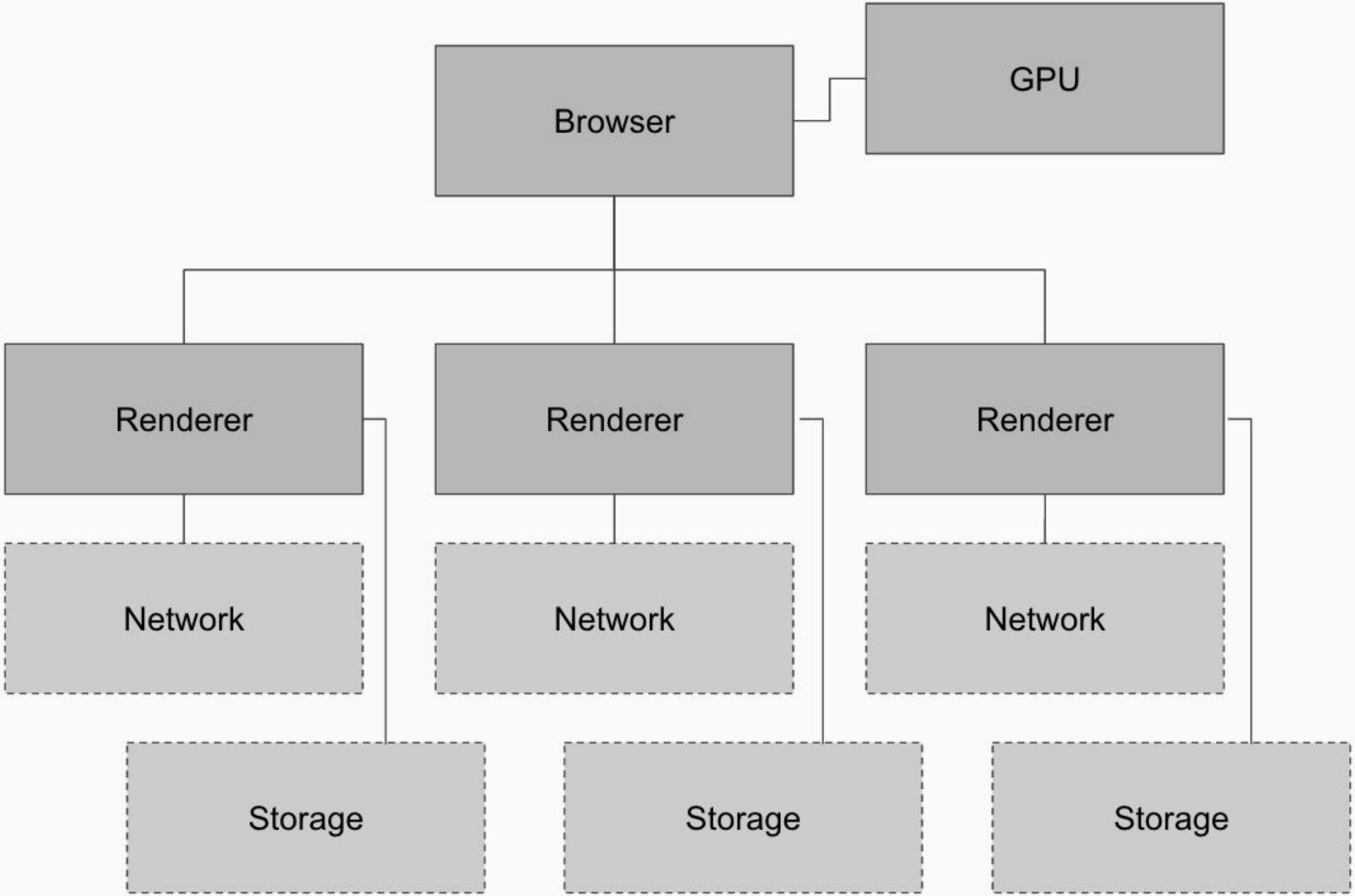
Renderer

Renderer


GPU
Network
Browser
Renderer
Renderer
Renderer


GPU
Browser
Renderer
Renderer
Renderer
Network
Network
Network


GPU
Browser
Storage
Renderer
Renderer
Renderer
Network
Network
Network


GPU
Browser
Renderer
Renderer
Renderer
Network
Network
Network
Storage
Storage
Storage

Moving Forward: Memory Safety

Investigating Safer Language Options

- Java/Kotlin on Android
- Swift on iOS/macOS
- Rust
- WebAssembly?

Migrating To Memory-Safe Languages

Not all-or-nothing! Thankfully!

We can focus on hot spots: areas of particularly large, soft, or easily-accessible attack surface.

Interoperability and overall complexity are huge concerns.

Learning curve, too.

Improving Memory-Unsafe Languages

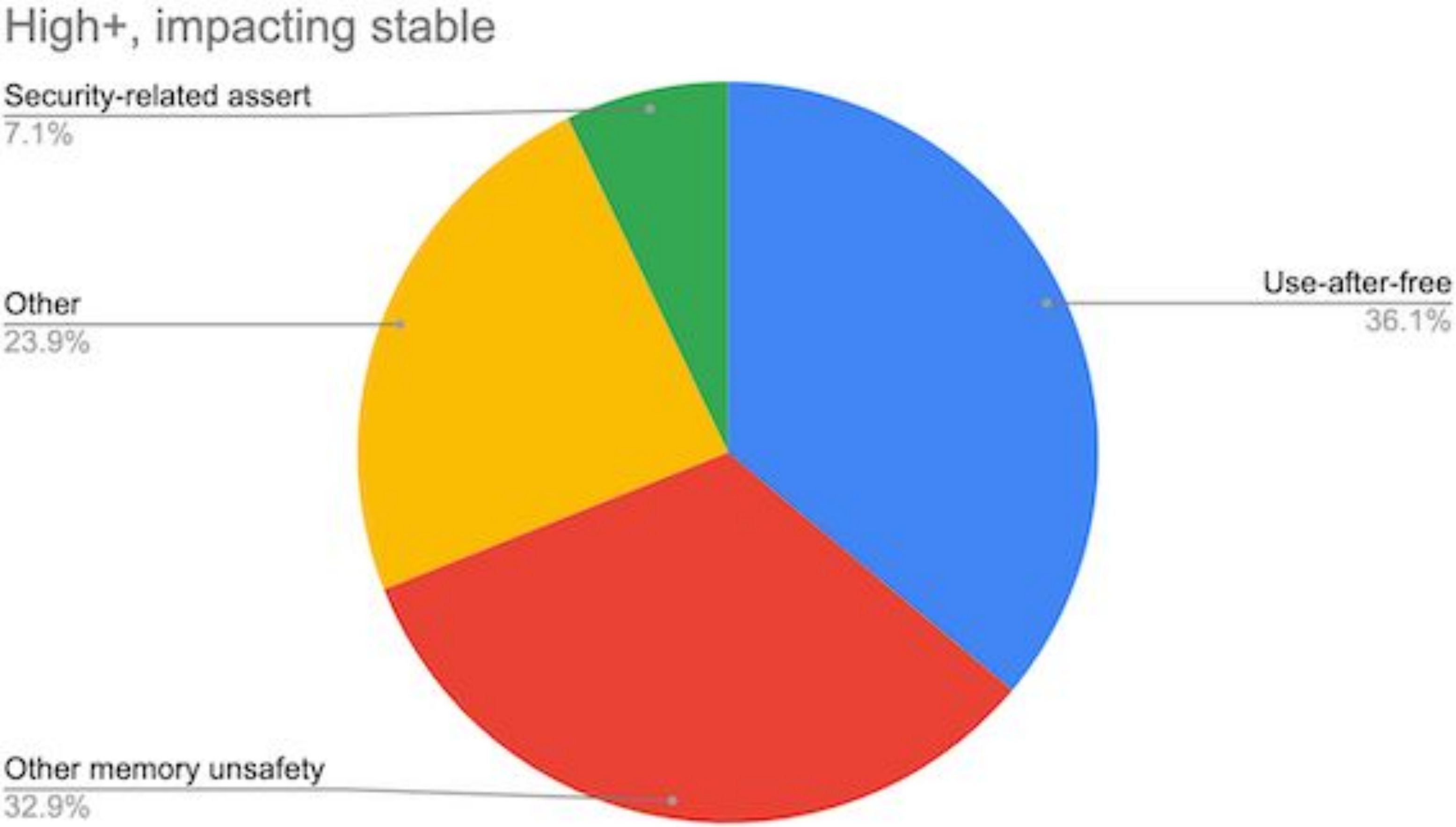
Smarter pointer types (MiraclePtr)

Garbage collection (expanding the use of Oilpan) and semi-GC (MiracleScan)

Defining undefined behavior

New hardware features (memory tagging, control flow integrity)

Implications



Roughly 70% of High+ Chromium vulnerabilities are memory unsafety

The Future

Sandboxing has given Chromium 10+ good years!

The next 10 require something more.

Questions?