

Retrofitting Fine Grain Isolation in the Firefox Renderer

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm,
Sorin Lerner, Hovav Shacham, Deian Stefan

UC San Diego

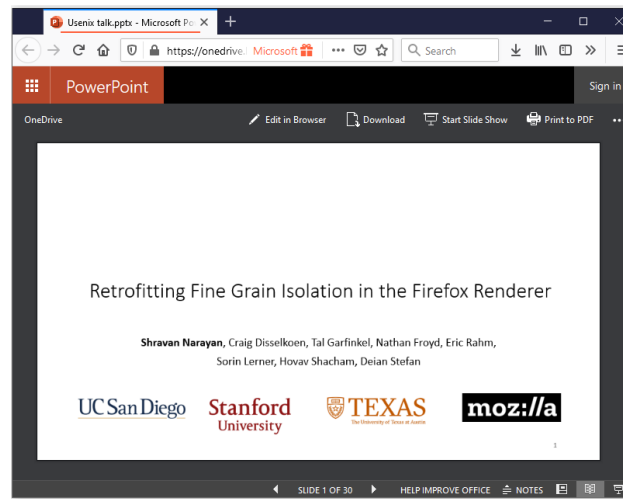
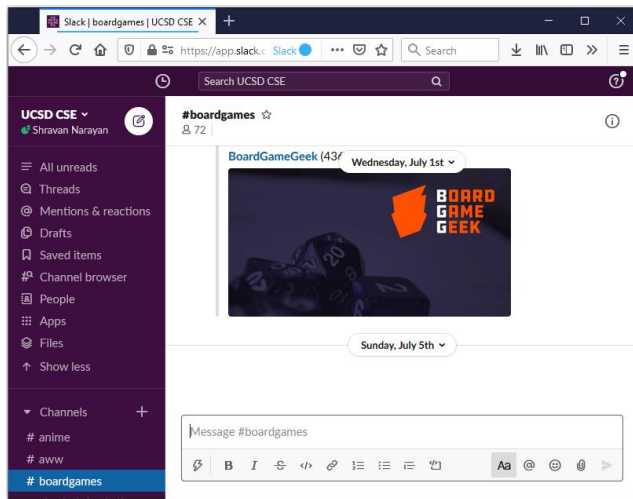
Stanford
University

 **TEXAS**
The University of Texas at Austin

moz://a

We use browsers for everything

Email, IM, conferences, meetings etc.



Third-party libraries make this possible

Browsers cannot implement every feature from scratch

Browsers use third-party libraries

- Used to render audio, video, images etc.

Large number of supported formats and libraries

- Images – JPEG, PNG, GIF, SVG, BMP, APNG, ICO, TIFF, WebP
- Video – H.264, VP8, VP9, Theora
- Audio – MP3, WAV, AAC, Vorbis, FLAC, Opus

Bugs in libraries can compromise browsers

CVE-2018-5146: Out of bounds memory write in libvorbis (2018 Pwn2Own)

Reporter Richard Zhu via Trend Micro's Zero Day Initiative

Impact critical

Description

An out of bounds memory write while processing Vorbis audio data was reported through the Pwn2Own contest.

References

[Bug 1446062](#)

JSC Exploits

Posted by Samuel Groß, Project Zero

In this post, we will take a look at the WebKit exploits used to gain an initial foothold onto the iOS device and stage the privilege escalation exploits. All exploits here achieve shellcode execution inside the sandboxed renderer process (WebContent) on iOS. Although Chrome on iOS would have also been vulnerable to these initial browser exploits, they were only used by the attacker to target Safari and iPhones.

After some general discussion, this post first provides a short walkthrough of each of the exploited WebKit bugs and how the attackers construct a memory read/write primitive from them, followed by an overview of the techniques used to gain shellcode execution and how they bypassed existing JIT code injection mitigations, namely the "bulletproof JIT".

Responding to Firefox 0-days in the wild



Philip Martin [Follow](#)

Aug 8, 2019 · 7 min read



On Thursday, May 30, over a dozen Coinbase employees received an email purporting to be from Gregory Harris, a Research Grants Administrator at the University of Cambridge. This email came from the legitimate Cambridge domain, contained no malicious elements, passed spam detection, and referenced the backgrounds of the recipients. Over the next couple weeks, similar emails were received. Nothing seemed amiss.

How do browsers deal with bugs in libraries?

Traditionally: Coarse-grain renderer isolation

- Goal: protect system from browser compromise
- Isolates the renderer (code handling untrusted HTML, images, JavaScript)

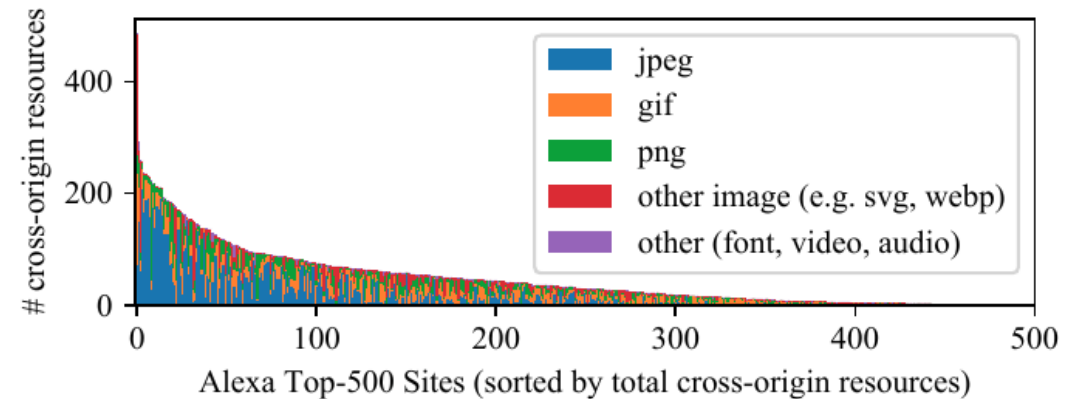
More recently: Site isolation

- Goal: protect one site from another
- Isolates different sites from each other
 - E.g., `*.google.com` is isolated from `*.zoom.us`

Why Site Isolation is not enough

Real sites rely on cross origin resources

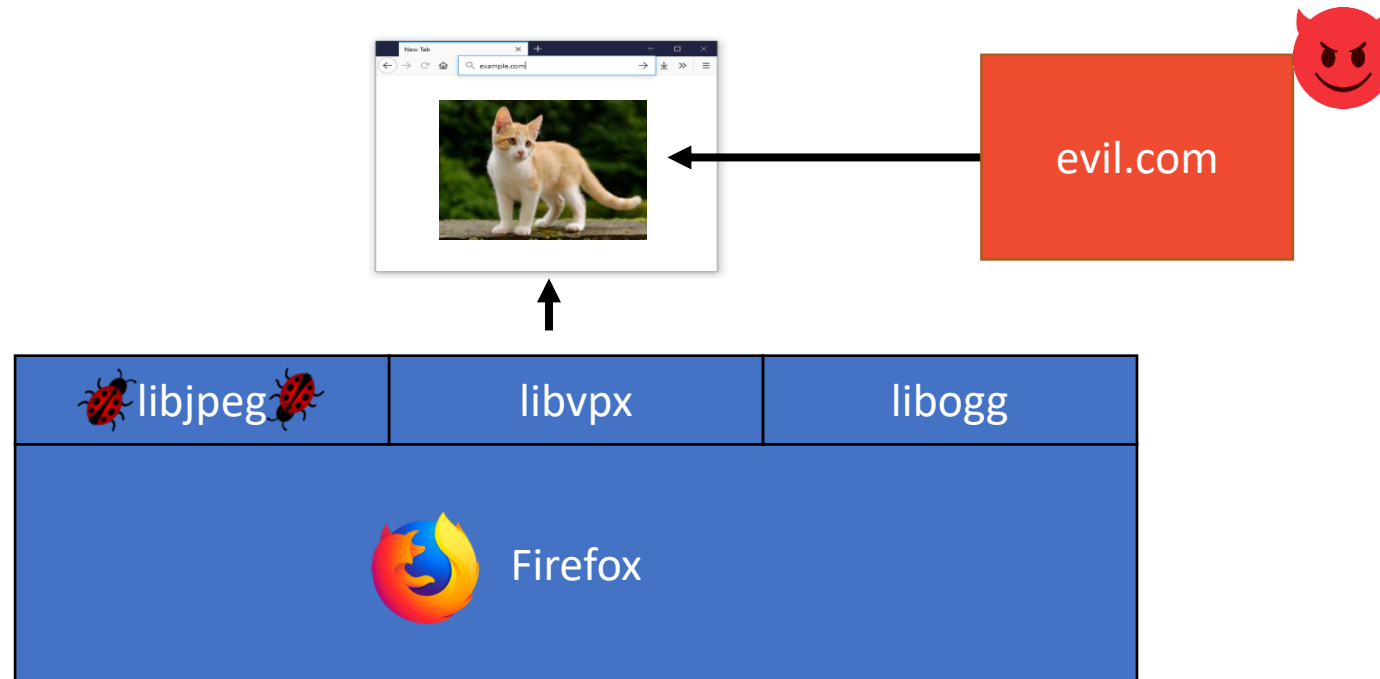
- 93% of sites load cross-origin media
- Lots of cross origin jpegs
- Bug in `libjpeg` \Rightarrow renderer compromise



Attacker may be able to host untrusted content on same origin

- Malicious media on Google Drive \Rightarrow compromised renderer
- Allows access victim's Drive files

We need fine grain isolation



Isolate/sandbox media libraries like `libjpeg`

- Bugs in `libjpeg` should not compromise the rest of Firefox

We know how to do this!

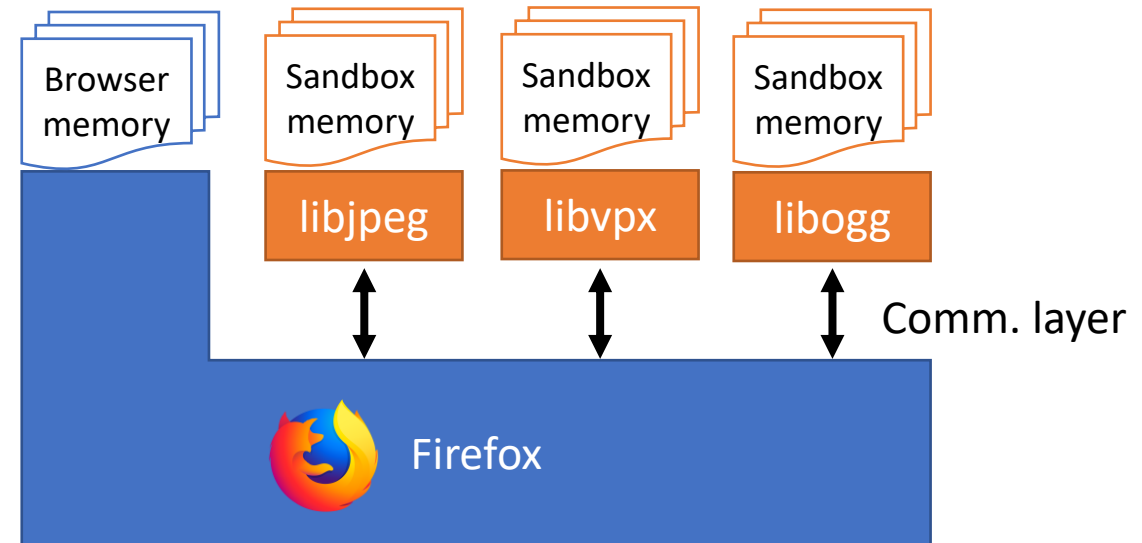
1. Pick an isolation/sandboxing mechanism

- Process isolation
- In-process: Native Client, WebAssembly

2. Put `libjpeg` in this sandbox

- `libjpeg` can only access sandbox memory

Done?



Isolation is not the only concern

Firefox code was written to trust `libjpeg`

- No sanitization of `libjpeg` data \Rightarrow renderer compromise

Isolation mechanism may introduce ABI differences

- Eg: not accounting for this \Rightarrow renderer compromise

Engineering challenges

- Difficult to disaggregate the tightly coupled data & control flow

```
void create_jpeg_parser() {  
  
    jpeg_decompress_struct jpeg_img;  
    jpeg_source_mgr        jpeg_input_source_mgr;  
  
    jpeg_create_decompress(&jpeg_img);  
    jpeg_img.src = &jpeg_input_source_mgr;  
    jpeg_img.src->fill_input_buffer = /* Set input bytes source */;
```

Now-untrusted jpeg initialized struct

```
    jpeg_read_header(&jpeg_img /* ... */);  
    uint32_t* outputBuffer = /* ... */;
```

```
    while (/* check for output lines */) {  
        uint32_t size = jpeg_img.output_width * jpeg_img.output_components;
```

```
        memcpy(outputBuffer, /* ... */, size);
```

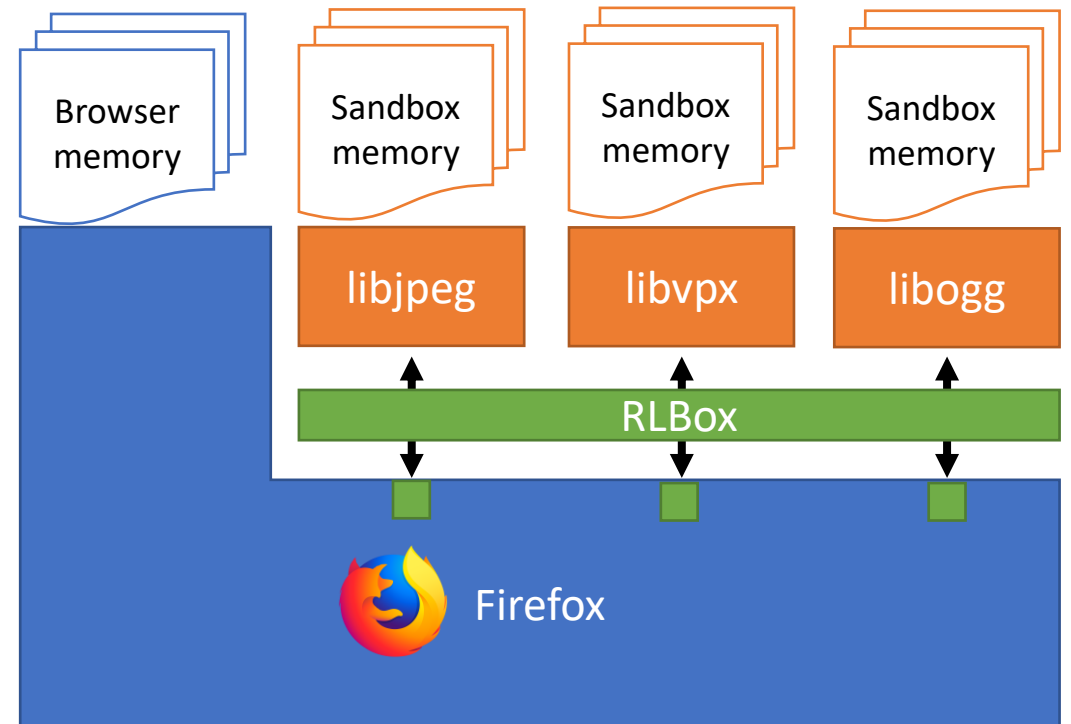
Using unchecked data from sandbox

```
    }  
}
```

RLBox

A C++ library that:

1. Abstracts isolation mechanism
 - Sandboxing with chosen isolation mechanism
 - Process, Native Client, WebAssembly, etc.
2. Mediates app-sandbox communication
 - APIs for control flow in/out of sandbox
 - `tainted` types for data flow in/out of sandbox



Marking data from the sandbox tainted...

1. Ensures potentially unsafe data is validated before use
2. Automates ABI conversions & certain validations
3. Enables incremental porting
4. Minimizes renderer code changes
5. Allows sharing data structures
 - Lazy data marshalling

```
void create_jpeg_parser() {

    jpeg_decompress_struct jpeg_img;
    jpeg_source_mgr      jpeg_input_source_mgr;

    jpeg_create_decompress(&jpeg_img);
    jpeg_img.src = &jpeg_input_source_mgr;

    jpeg_img.src->fill_input_buffer = /* Set input bytes source */;

    jpeg_read_header(&jpeg_img /* ... */);
    uint32_t* outputBuffer = /* ... */;

    while (/* check for output lines */) {
        uint32_t size = jpeg_img.output_width * jpeg_img.output_components;

        memcpy(outputBuffer, /* ... */, size);
    }
}
```

```
void create_jpeg_parser() {

    jpeg_decompress_struct jpeg_img;
    jpeg_source_mgr      jpeg_input_source_mgr;

    jpeg_create_decompress(&jpeg_img);
    jpeg_img.src = &jpeg_input_source_mgr;

    jpeg_img.src->fill_input_buffer = /* Set input bytes source */;

    jpeg_read_header(&jpeg_img /* ... */);
    uint32_t* outputBuffer = /* ... */;

    while (/* check for output lines */) {
        uint32_t size = jpeg_img.output_width * jpeg_img.output_components;

        memcpy(outputBuffer, /* ... */, size);
    }
}
```

```
void create_jpeg_parser() {
```

```
    auto sandbox = rlbbox::create_sandbox<wasm>();
```

```
    jpeg_decompress_struct jpeg_img;
```

```
    jpeg_source_mgr        jpeg_input_source_mgr;
```

```
    jpeg_create_decompress(&jpeg_img);
```

```
    jpeg_img.src = &jpeg_input_source_mgr;
```

```
    jpeg_img.src->fill_input_buffer = /* Set input bytes source */;
```

```
    jpeg_read_header(&jpeg_img /* ... */);
```

```
    uint32_t* outputBuffer = /* ... */;
```

```
    while (/* check for output lines */) {
```

```
        uint32_t size = jpeg_img.output_width * jpeg_img.output_components;
```

```
        memcpy(outputBuffer, /* ... */, size);
```

```
    }
```

```
}
```

Invoke jpeg functions via RLBox

```
void create_jpeg_parser() {
```

```
    auto sandbox = rlbbox::create_sandbox<wasm>();
```

```
    jpeg_decompress_struct jpeg_img;
```

```
    jpeg_source_mgr        jpeg_input_source_mgr;
```

```
    sandbox.invoke(jpeg_create_decompress, &jpeg_img);
```

```
    jpeg_img.src = &jpeg_input_source_mgr;
```

```
    jpeg_img.src->fill_input_buffer = /* Set input bytes source */;
```

```
    jpeg_read_header(&jpeg_img /* ... */);
```

```
    uint32_t* outputBuffer = /* ... */;
```

```
    while (/* check for output lines */) {
```

```
        uint32_t size = jpeg_img.output_width * jpeg_img.output_components;
```

```
        memcpy(outputBuffer, /* ... */, size);
```

```
    }
```

```
}
```

Invoke jpeg functions via RLBox


```
void create_jpeg_parser() {
```

```
    auto sandbox = r1box::create_sandbox<wasm>();
```

```
    jpeg_decompress_struct jpeg_img;
```

```
    jpeg_source_mgr        jpeg_input_source_mgr;
```

```
    sandbox.invoke(jpeg_create_decompress, &jpeg_img);
```

```
    jpeg_img.src = &jpeg_input_source_mgr;
```

```
    jpeg_img.src->fill_input_buffer = /* Set input bytes source */;
```

```
    jpeg_read_header(&jpeg_img /* ... */);
```

```
    uint32_t* outputBuffer = /* ... */;
```

```
    while (/* check for output lines */) {
```

```
        uint32_t size = jpeg_img.output_width * jpeg_img.output_components;
```

```
        memcpy(outputBuffer, /* ... */, size);
```

```
    }
```

```
}
```

Expected: tainted<jpeg_decompress_struct*>

Compiles?



```

void create_jpeg_parser() {

    auto sandbox = r1box::create_sandbox<wasm>();
    tainted<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();
    jpeg_source_mgr          jpeg_input_source_mgr;

    sandbox.invoke(jpeg_create_decompress, &jpeg_img);
    jpeg_img.src = &jpeg_input_source_mgr;

    jpeg_img.src->fill_input_buffer = /* Set input bytes source */;

    jpeg_read_header(&jpeg_img /* ... */);
    uint32_t* outputBuffer = /* ... */;

    while (/* check for output lines */) {
        uint32_t size = jpeg_img.output_width * jpeg_img.output_components;

        memcpy(outputBuffer, /* ... */, size);
    }
}

```

```
void create_jpeg_parser() {
```

```
    auto sandbox = r1box::create_sandbox<wasm>();
```

```
    tainted<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();
```

```
    jpeg_source_mgr          jpeg_input_source_mgr;
```

```
    sandbox.invoke(jpeg_create_decompress, p_jpeg_img);
```

```
    p_jpeg_img->src = &jpeg_input_source_mgr;
```

Expected: tainted<jpeg_source_mgr*>

```
    jpeg_img.src->fill_input_buffer = /* Set input bytes source */;
```

```
    jpeg_read_header(&jpeg_img /* ... */);
```

```
    uint32_t* outputBuffer = /* ... */;
```

```
    while (/* check for output lines */) {
```

```
        uint32_t size = jpeg_img.output_width * jpeg_img.output_components;
```

```
        memcpy(outputBuffer, /* ... */, size);
```

```
    }
```

```
}
```

Compiles?



```

void create_jpeg_parser() {

    auto sandbox = ribox::create_sandbox<wasm>();
    tainted<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();
    tainted<jpeg_source_mgr*> p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

    sandbox.invoke(jpeg_create_decompress, p_jpeg_img);
    p_jpeg_img->src = p_jpeg_input_source_mgr;

    p_jpeg_img->src->fill_input_buffer = /* Set input bytes source */;

    sandbox.invoke(jpeg_read_header, p_jpeg_img /* ... */);
    uint32_t* outputBuffer = /* ... */;

    while (/* check for output lines */) {
        tainted<uint32_t> size = p_jpeg_img->output_width * p_jpeg_img->output_components;

        memcpy(outputBuffer, /* ... */, size);
    }
}

```

1. RLBox adjusts for ABI differences

2. RLBox bounds checks this dereference

3. size is tainted

```

void create_jpeg_parser() {

    auto sandbox = r1box::create_sandbox<wasm>();
    tainted<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();
    tainted<jpeg_source_mgr*> p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

    sandbox.invoke(jpeg_create_decompress, p_jpeg_img);
    p_jpeg_img->src = p_jpeg_input_source_mgr;

    p_jpeg_img->src->fill_input_buffer = /* Set input bytes source */;

    sandbox.invoke(jpeg_read_header, p_jpeg_img /* ... */);
    uint32_t* outputBuffer = /* ... */;

    while (/* check for output lines */) {
        tainted<uint32_t> size = p_jpeg_img->output_width * p_jpeg_img->output_components;

        memcpy(outputBuffer, /* ... */, size);
    }
}

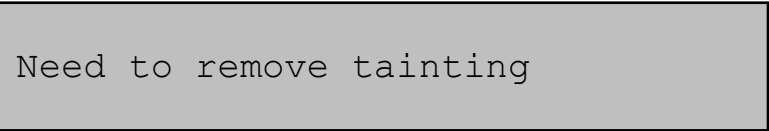
```

Expected: uint32_t
Got: tainted<uint32_t>

Compiles?



```
void create_jpeg_parser() {  
  
    auto sandbox = r1box::create_sandbox<wasm>();  
    tainted<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();  
    tainted<jpeg_source_mgr*>          p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();  
  
    sandbox.invoke(jpeg_create_decompress, p_jpeg_img);  
    p_jpeg_img->src = p_jpeg_input_source_mgr;  
  
    p_jpeg_img->src->fill_input_buffer = /* Set input bytes source */;  
  
    sandbox.invoke(jpeg_read_header, p_jpeg_img /* ... */);  
    uint32_t* outputBuffer = /* ... */;  
  
    while (/* check for output lines */) {  
        tainted<uint32_t> size = p_jpeg_img->output_width * p_jpeg_img->output_components;  
  
        memcpy(outputBuffer, /* ... */, size);  
    }  
}
```



Need to remove tainting

```

void create_jpeg_parser() {

    auto sandbox = r1box::create_sandbox<wasm>();
    tainted<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();
    tainted<jpeg_source_mgr*> p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

    sandbox.invoke(jpeg_create_decompress, p_jpeg_img);
    p_jpeg_img->src = p_jpeg_input_source_mgr;

    p_jpeg_img->src->fill_input_buffer = /* Set input bytes source */;

    sandbox.invoke(jpeg_read_header, p_jpeg_img /* ... */);
    uint32_t* outputBuffer = /* ... */;

    while (/* check for output lines */) {
        uint32_t size = (p_jpeg_img->output_width * p_jpeg_img->output_components).copy_and_verify(
            [](uint32_t val) -> uint32_t {
                ...
            });
        memcpy(outputBuffer, /* ... */, size);
    }
}

```

```

void create_jpeg_parser() {

    auto sandbox = r1box::create_sandbox<wasm>();
    tainted<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();
    tainted<jpeg_source_mgr*> p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

    sandbox.invoke(jpeg_create_decompress, p_jpeg_img);
    p_jpeg_img->src = p_jpeg_input_source_mgr;

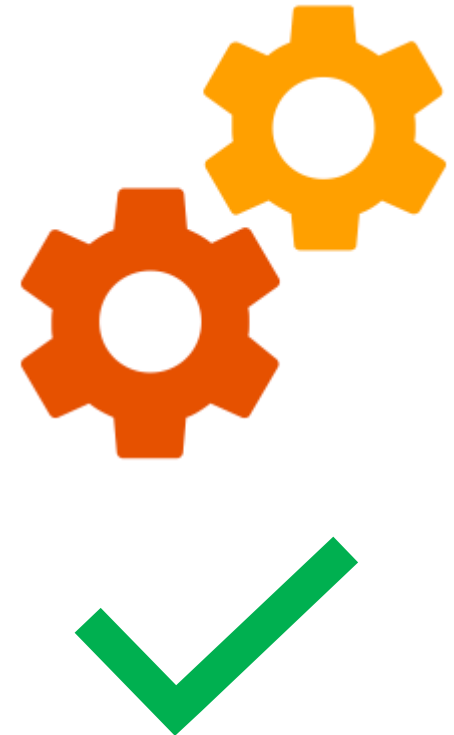
    p_jpeg_img->src->fill_input_buffer = /* Set input bytes source */;

    sandbox.invoke(jpeg_read_header, p_jpeg_img /* ... */);
    uint32_t* outputBuffer = /* ... */;

    while (/* check for output lines */) {
        uint32_t size = (p_jpeg_img->output_width * p_jpeg_img->output_components).copy_and_verify(
            [](uint32_t val) -> uint32_t {
                assert(val <= outputBufferSize);
                return val;
            });
        memcpy(outputBuffer, /* ... */, size);
    }
}

```

Compiles?



How well does this work in a real codebase?

We sandboxed different kinds of libraries in Firefox

- Image libraries – `libjpeg`, `libpng`
- Video libraries – `libtheora`, `libvpx`
- Audio library – `libogg`
- Compression library – `zlib`

We evaluate RLBox on several dimensions. In this talk:

- Developer effort & automation
- Performance overhead

Developer effort (Takeaway)

On average, sandboxing a library takes only a few days

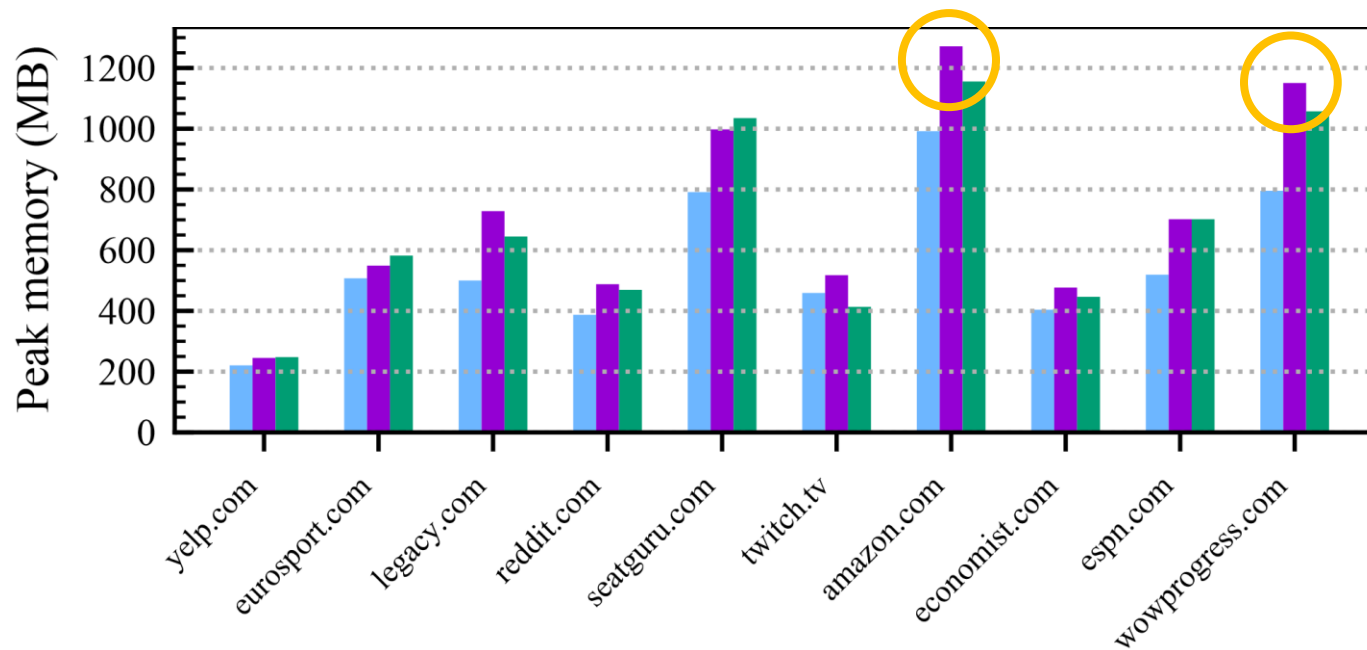
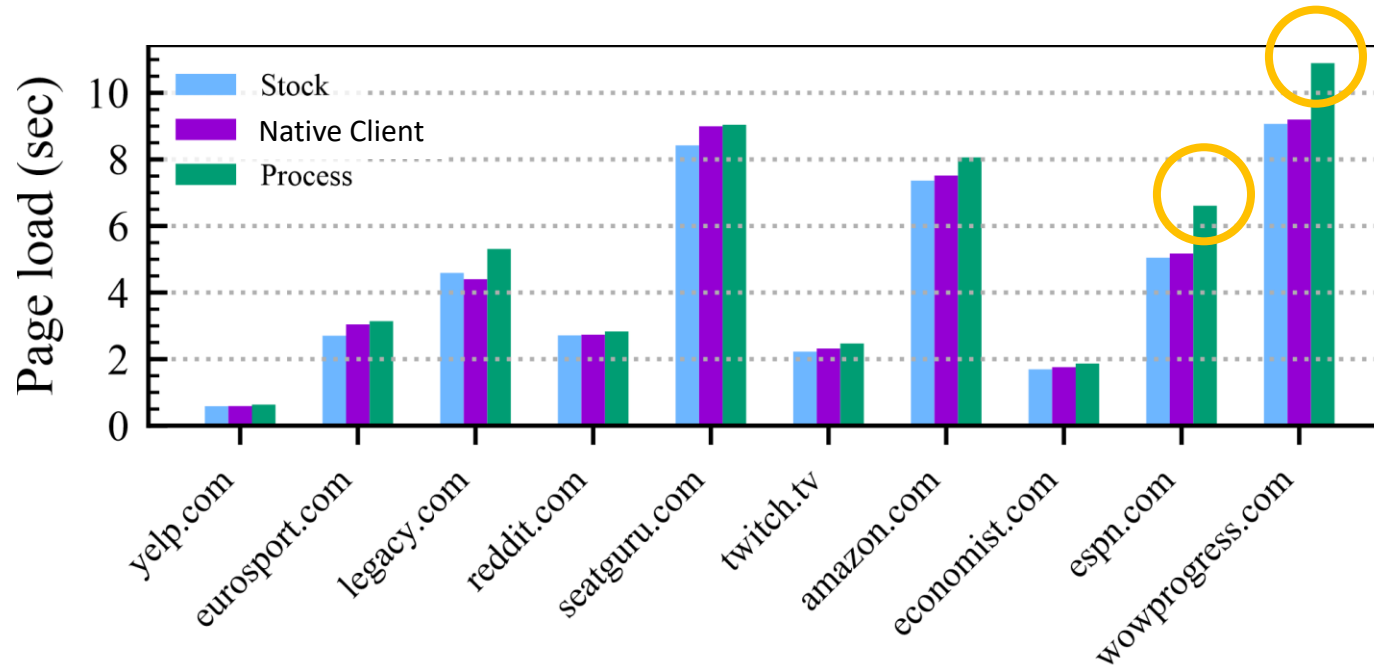
RLBox automation

- Bounds checks: 8-64 (average: 23)
- Nested ABI conversions: 5-17 (average: 7)

Locations that need validators: 2-51 (average: 17)

- Validators are between 2-4 lines of code

Performance impact (Takeaway)





moz://a

HACKS

[Securing Firefox with WebAssembly](#)



By [Nathan Froyd](#)

Posted on [February 25, 2020](#) in [Featured Article](#), [Firefox](#), [Rust](#), [Security](#), and [WebAssembly](#)

Protecting the security and privacy of individuals is a [central tenet](#) of Mozilla's mission, and so we constantly endeavor to make our users safer online. With a

...

So today, we're adding a third approach to our arsenal. [RLBox](#), a new sandboxing technology developed by researchers at the University of California, San Diego, the University of Texas, Austin, and Stanford University, allows us to quickly and efficiently convert existing Firefox components to run inside a

<https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/>