RLBox: secure sandboxing for buggy/unsafe application components

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm,

Sorin Lerner, Hovav Shacham, Deian Stefan

UC San Diego

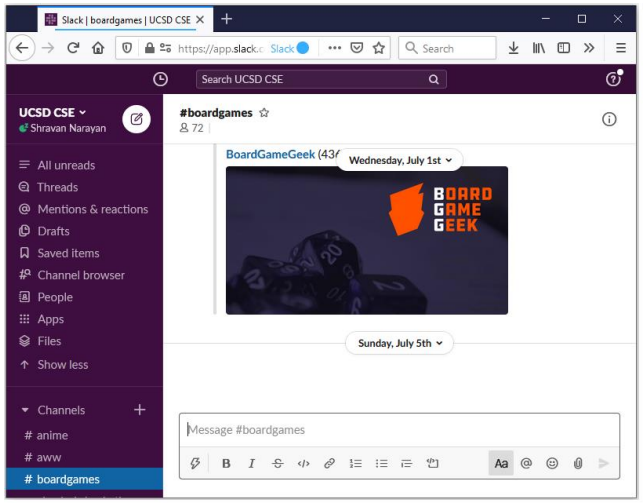
Stanford University

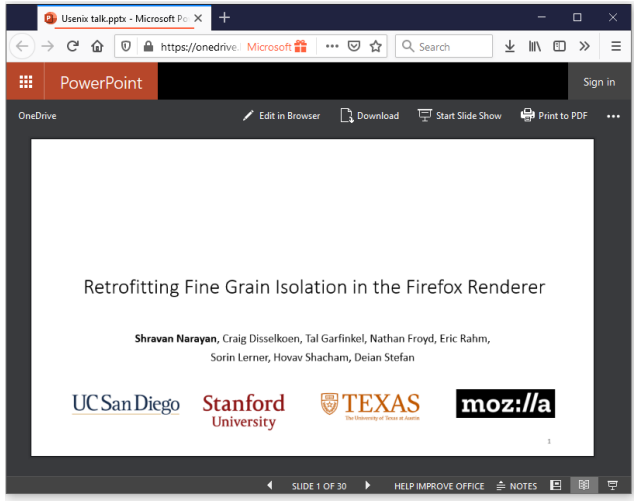
TEXAS
The University of Texas at Austin

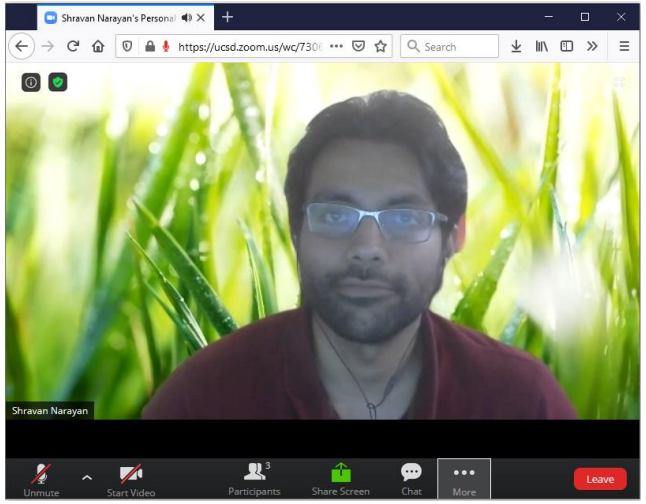
moz://a

Running example: Browsers

We use browsers for everything







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

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)

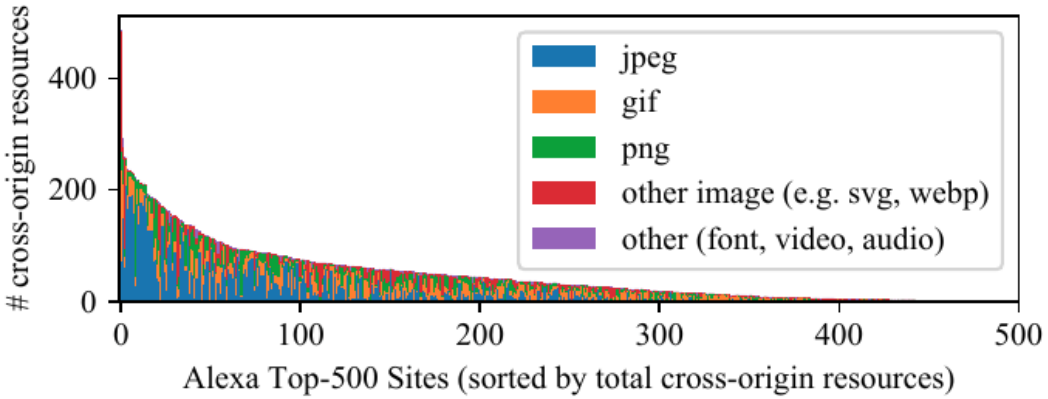
More recently: Site isolation

- 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

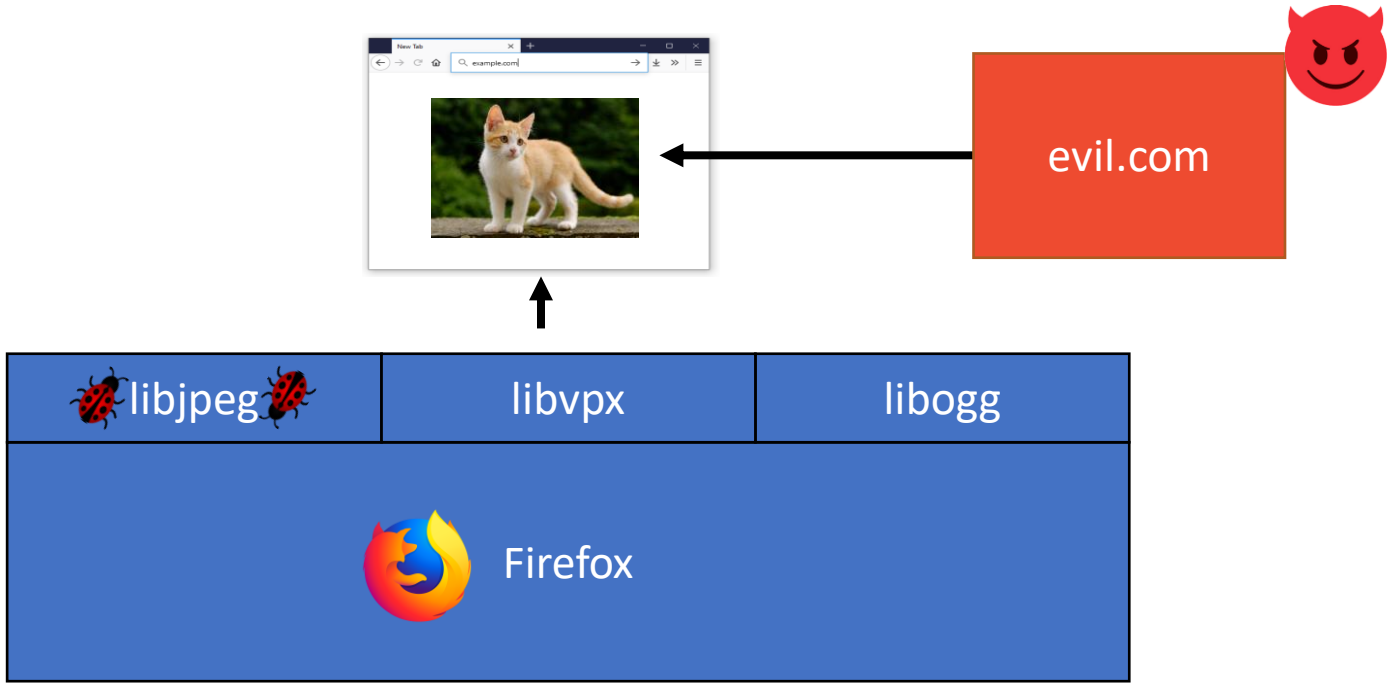
- 93% of sites load cross-origin media
- Lots of cross origin jpegs
- Bug in `libjpeg` ⇒ renderer compromise



Attacker may be able to host untrusted content on same origin

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

We need fine grain isolation



Isolate/sandbox media libraries like `libjpeg`

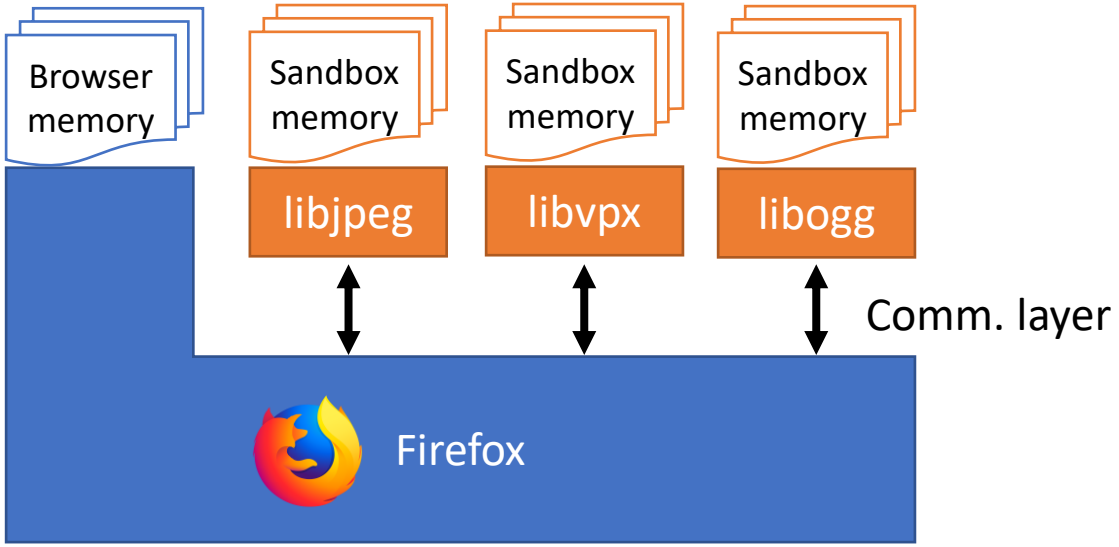
- Bugs in `libjpeg` should not compromise the rest of Firefox

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 ⇒ renderer compromise

Isolation mechanism may introduce ABI differences

- Eg: not accounting for this ⇒ 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 */;

  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);
  }
}
```

Now-untrusted jpeg initialized struct

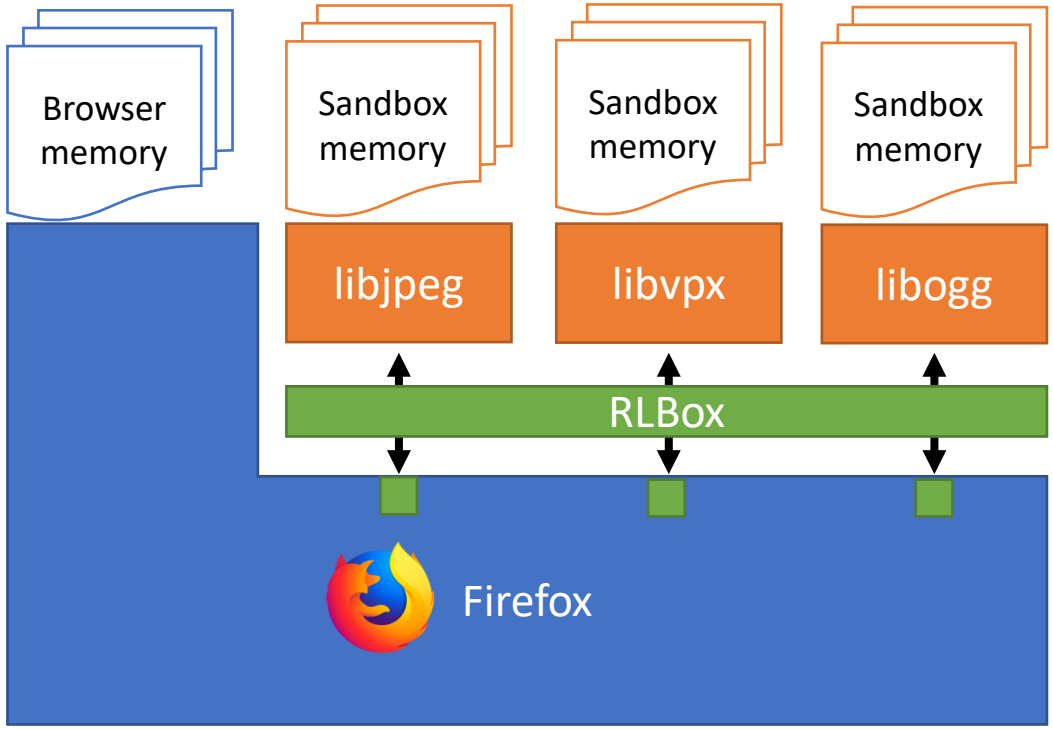
Using unchecked data from sandbox

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 = rlbox::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 = rlbox::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 = rlbox::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 = rlbox::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 = rlbox::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;

  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_source_mgr*>

Compiles?

```
void create_jpeg_parser() {

  auto sandbox = rlbox::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;

  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);
  }
}
```

Undefined variable: jpeg_img

Compiles?

```
void create_jpeg_parser() {

  auto sandbox = rlbox::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;
  jpeg_decompress_struct& jpeg_img = *p_jpeg_img.UNSAFE_unverified();
  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 = rlbox::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;
  jpeg_decompress_struct& jpeg_img = *p_jpeg_img.UNSAFE_unverified();
  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 = rlbox::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

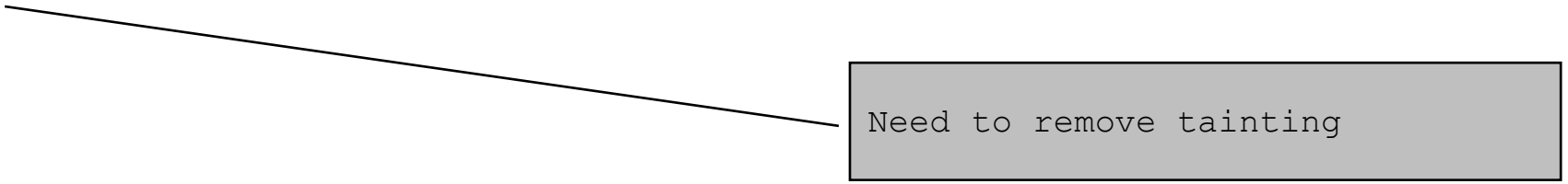
3. size is tainted

```
void create_jpeg_parser() {

  auto sandbox = rlbox::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);
  }
}
```

Compiles?

```
Expected: uint32_t
Got: tainted<uint32_t>
```

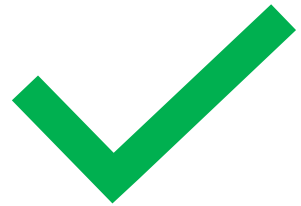
```
void create_jpeg_parser() {

  auto sandbox = rlbox::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);
  }
}
```



```
void create_jpeg_parser() {

  auto sandbox = rlbox::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 = rlbox::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

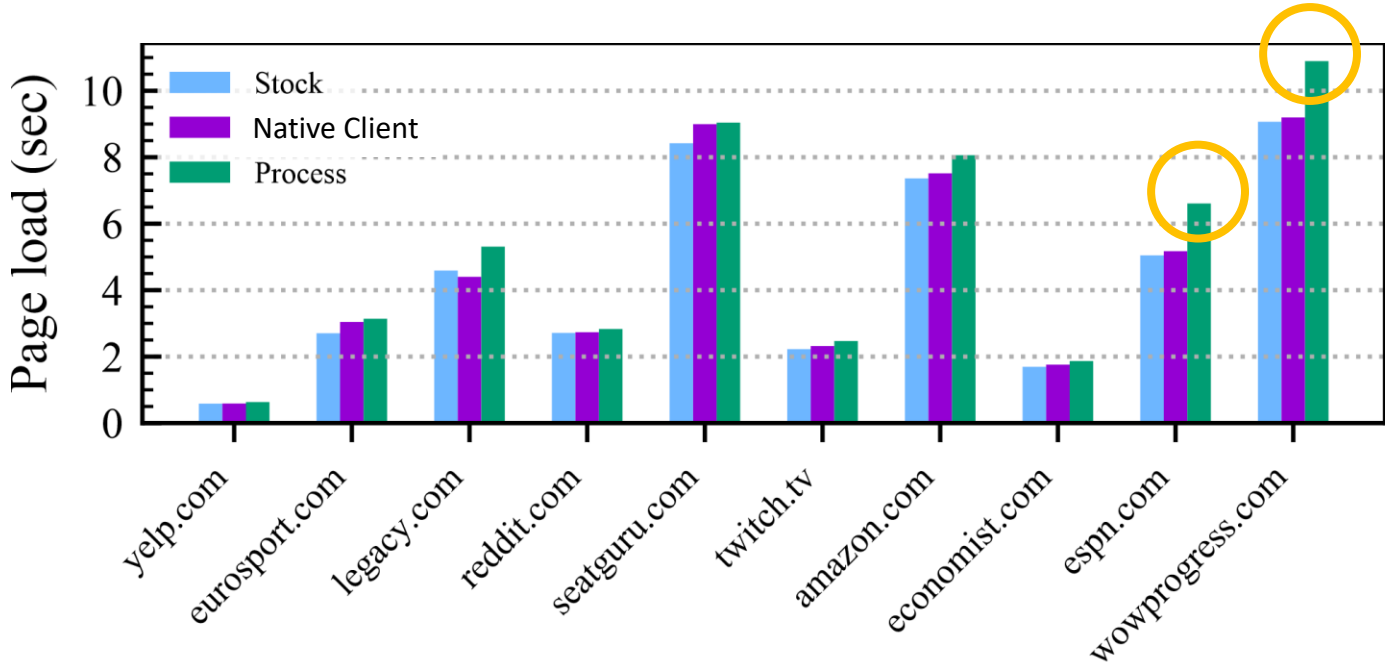
RLBox automation

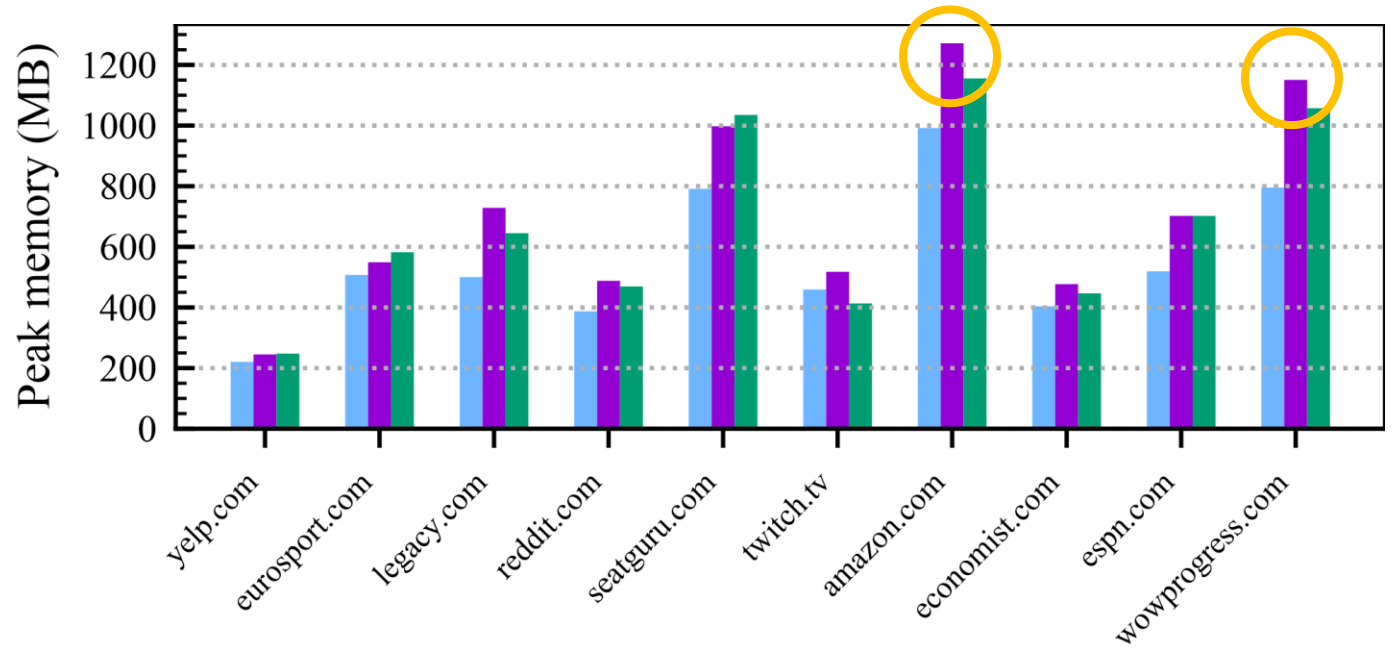
- Bounds checks: 8-64 (average: 23)
- Nested ABI conversions: 5-17 (average: 7)

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

- Validators are between 2-4 lines of code

Performance impact (Takeaway)





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/

Lots more in the paper

- Trade-offs in sandbox grain vs security
  - Fresh sandbox per image does not scale
- Porting
  - Handling callbacks, threading & longjmp
  - Validator patterns
- Implementation
  - RLBox internals, compiler toolchains etc.
  - Optimizations for NaCl, Process, Wasm
- Modifications needed to ship RLBox
  - Fully automated ABI conversion
- Benchmarking
  - Benchmarks on 6 sandboxed libraries
  - Measuring / addressing scalability
- Using RLBox in other systems
  - Apache web server
  - Node.js applications

Backup

Developer effort to use RLBox

	Task	JPEG Decoder	PNG Decoder	GZIP Decompress	Theora Decoder	VPX Decoder	OGG-Vorbis Decoder
Effort saved by RLBox automation	Generated marshaling code	133 LOC	278 LOC	38 LOC	39 LOC	60 LOC	59 LOC
	Automatic pointer swizzles for function calls	30	96	5	36	46	34
	Automatic nested pointer swizzles	17	5	6	8	9	5
	Automatic pointer bounds checks	64 checks	25 checks	8 checks	12 checks	15 checks	14 checks
	Number of validator sites found	28	51	10	5	2	4
Manual effort	Number of person-days porting to RLBox	–	–	1 day	3 days	3 days	2 days
	Application LOC before/after port	720 / 1058	847 / 1317	649 / 757	220 / 297	286 / 368	328 / 395
	Number of unique validators needed	11	14	3	3	2	2
	Average LOC of validators	3 LOC	4 LOC	2 LOC	3 LOC	2 LOC	2 LOC

Figure 3: Manual effort required to retrofit Firefox with fine grain isolation, including the effort saved by RLBox's automation. We do not report the number of days it took to port the JPEG and PNG decoders since we ported them in sync with building RLBox.

Sandboxing security policies

Sandboxing some components may not improve security

- Eg: Sandboxing the JavaScript engine

Sandboxing granularity affect scalability and security

- Sandbox per image/media does not scale – need to share sandboxes
- Unique sandbox per `<renderer ,library, content-origin>`
- Need even finer grain for libraries like `zlib`
  - `<renderer ,library, content-origin, content-type>`

Performance overhead of RLBox

Cost of RLBox's runtime checks: << 1% overhead

Mechanism	Function call overhead	Sandboxed code Overhead
None	~10 ns	0%
SFI with Native Client	220 ns	22%
Processes	7400 ns	0%
Processes using multicores & spinlocks	470 ns	0%
WebAssembly	< 100 ns	90%

Optimizations on isolation mechanisms

Native Client

- Hand-written SIMD support is limited
- Modified the Nasm compiler to produce NaCl compatible SIMD code

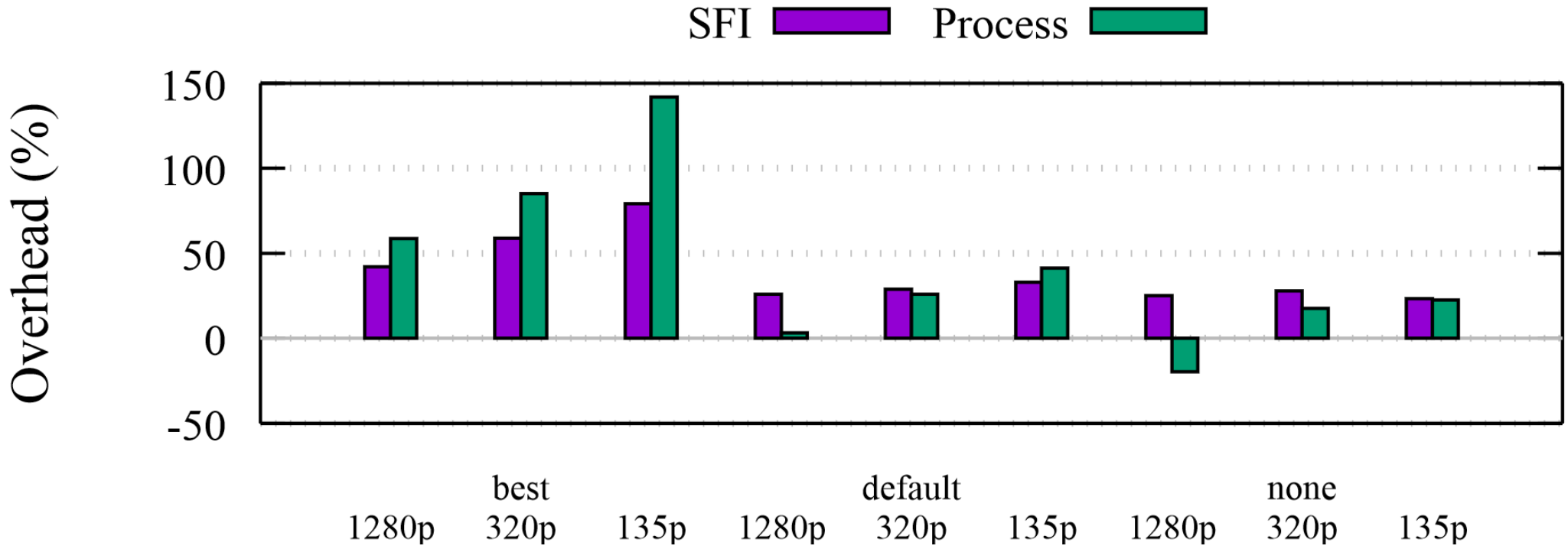
Process isolation

- Full process context switch is slow
- Optimization: Pin 2 processes on separate cores + use spinlocks on hotpath

Webassembly

- Example used in font rendering – lot of transitions
- Optimize transitions – full separate paper
- Move more code *into* the sandbox

Evaluation: Micro benchmarks



JPEG rendering overhead

Evaluation: Scalability