Tutorial: Using RLBox to sandbox unsafe C code

Shravan Narayan, Craig Disselkoen, Deian Stefan

UC San Diego

Before we start

The interactive part of this tutorial will need:

1. CMake build system

2. C++ compiler with C++ 17 support

This tutorial

1. Brief introduction to RLBox (20 mins)
2. Sandboxing a toy library (50 mins)
3. Preview of sandboxing in Firefox (10 mins)

Attackers are exploiting third party libraries

From Pearl to Pegasus

Bahraini Government Hacks Activists with NSO Group Zero-Click iPhone Exploits

By Bill Marczak, Ali Abdulemam[1], Noura Al-Jizawi, Siena Anstis, Kristin Berdan, John Scott-Railton, and Ron Deibert

[1] Red Line for Gulf

August 24, 2021

Phone logs show that (at least some of) the iOS 13.x and 14.x zero-click exploits deployed by NSO Group involved ImageIO, specifically the parsing JPEG and GIF images. ImageIO has had more than a dozen high-severity bugs reported against it in 2021.

Available for: iPhone 5s, iPhone 6, iPhone 6 Plus, iPad Air, iPad mini 2, iPad mini 3, and iPod touch (6th generation)

Impact: Processing a maliciously crafted PDF may lead to arbitrary code execution. Apple is aware of a report that this issue may have been actively exploited.

Description: An integer overflow was addressed with improved input validation.

CVE-2021-30860: The Citizen Lab

Issue 2161: QT: out-of-bounds read in TIFF processing

Reported by natashenka@google.com on Tue, Feb 23, 2021, 4:08 PM PST Project Member

The QImageReader class can read out-of-bounds when converting a specially-crafted TIFF file

---

0-days In-the-Wild Root Cause Analyses Tracking Sheet Contributing About

CVE-2020-15999: FreeType Heap Buffer Overflow in Load_SBit_Png

Sergei Glazunov, Project Zero (Originally posted on Project Zero blog 2021-02-04)

Issue 1196480: Security: Multiple Bugs in WebP

Reported by awhalley@google.com on Tue, Apr 6, 2021, 5:21 PM PDT Project Member

Comment 12 by cthomp@chromium.org on Wed, Apr 7, 2021, 12:22 PM PDT Project Member

I've filed child bugs to track each of the four security issues identified in the report:

(1) ~~Issue 1196773~~: Security: heap-use-after-free in libwebp ConvertBGRAToRGB_SSE41
(2) ~~Issue 1196775~~: Security: heap-buffer-overflow in libwebp PlanarTo24b_SSE41
(3) ~~Issue 1196777~~: Security: heap-buffer-overflow in libwebp VP8YuvToRgb
(4) ~~Issue 1196778~~: Security: heap-buffer-overflow in libwebp UpsampleRgbLinePair_SSE41

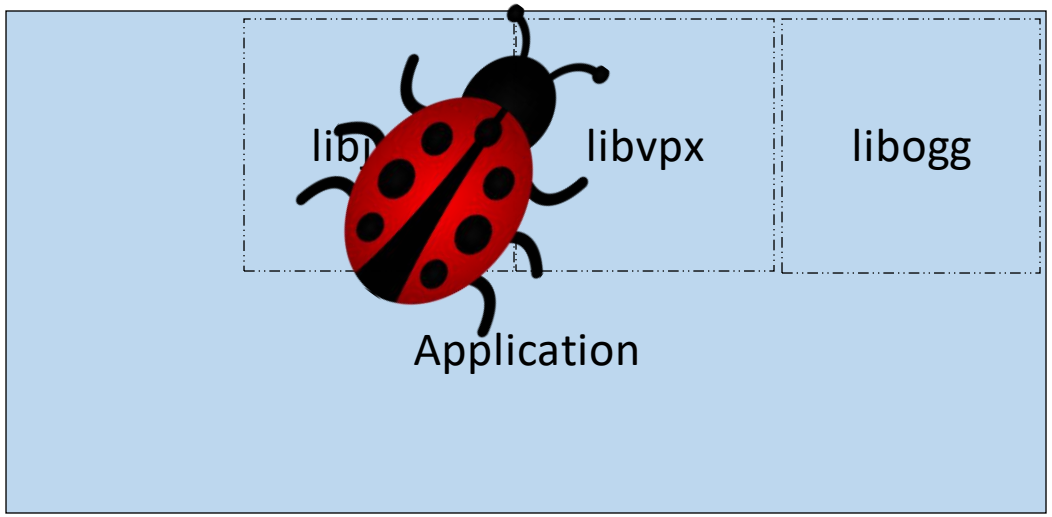
Vulnerability Details : CVE-2021-37972

Out of bounds read in libjpeg-turbo in Google Chrome prior to 94.0.4606.54 allowed a remote attacker to potentially exploit heap corruption via a crafted HTML page.

Publish Date : 2021-10-08 Last Update Date : 2021-10-10

The fundamental problem

Memory safety bug in a third-party lib => compromised an entire app



How do we fix this?

Find all bugs?

- We clearly can’t

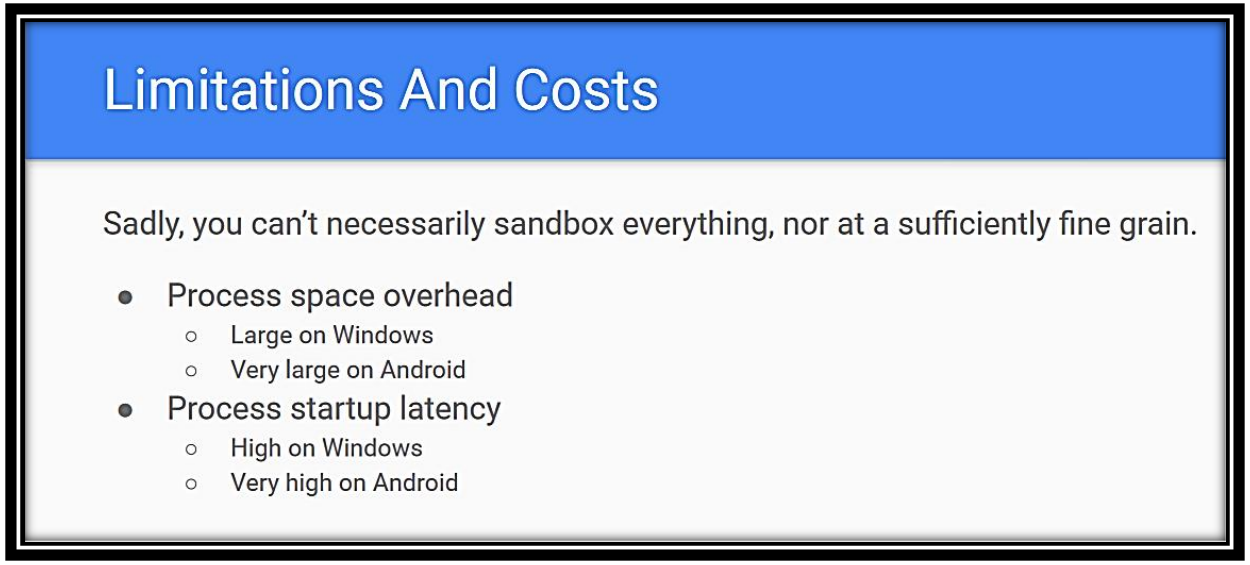
Rewrite everything in Rust?

- It’s too expensive

Use separate processes?

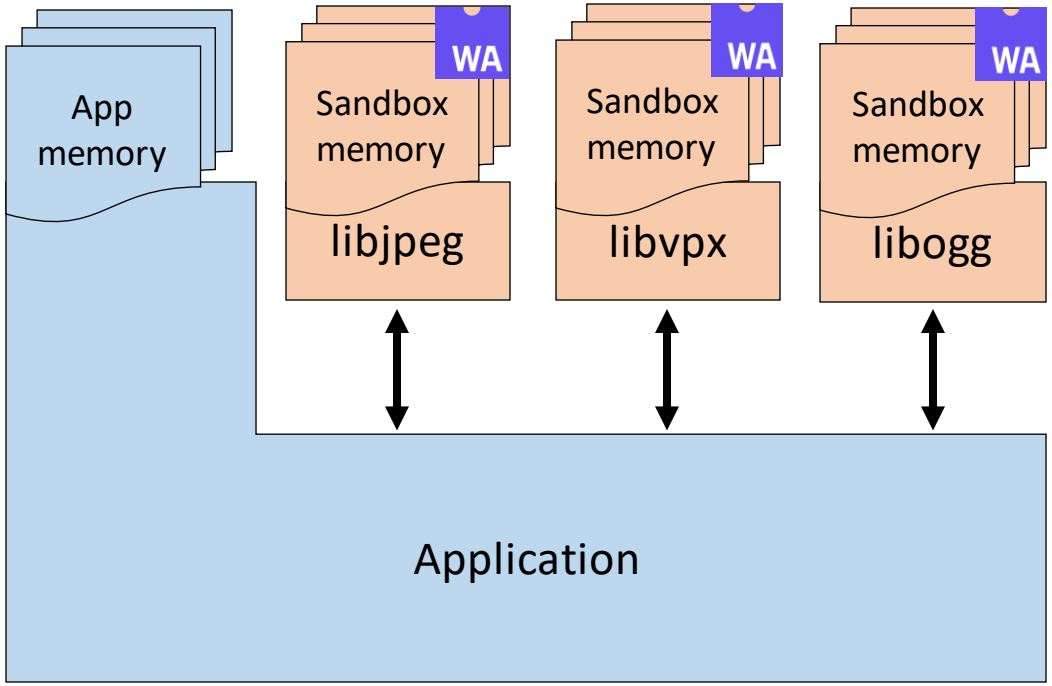
- It’s too expensive

Usenix Enigma 2021: Chris Palmer, Google Chrome Security



Our approach: Isolate libs in the same process

Libraries/components have their own memory



Challenge: Retrofitting library isolation

Retrofitting isolation is tricky!

Applications trust their libraries

- They don’t sanitize data from libraries
- Buggy library returns malformed data ⇒ application compromise

Libraries and applications are tightly coupled

- Figuring where to sanitize data and control flow is difficult

Isolation mechanisms like Wasm introduces ABI differences

- Wasm has 32-bit pointers
- Not accounting for this ⇒ application compromise

Let’s look at an example

```
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);
  }
}
```

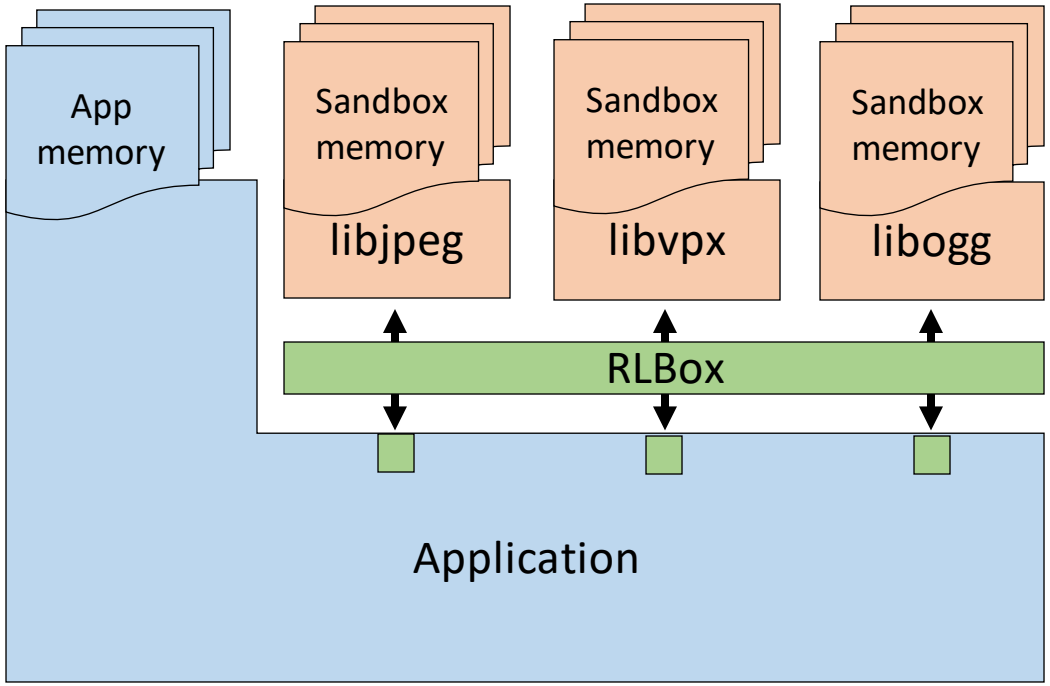
Now-untrusted jpeg initialized struct

Using unchecked data from sandbox

RLBox

A C++ library that:

1. Abstracts isolation mechanism
   - WebAssembly, Native Client etc.
2. Mediates app-sandbox communication
   - APIs to invoke sandboxed functions
   - Shared data is marked `tainted`



Deployed in production

Use WebAssembly to sandbox untrusted libraries in-process

What does RLBox give us?

1. Ensures `tainted` data is validated before use
2. Enables incremental porting
3. Automates ABI conversions & certain validations

```
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() {
  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();
  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() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

  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() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

  jpeg_decompress_struct jpeg_img;

  jpeg_source_mgr        jpeg_input_source_mgr;

  sandbox_invoke(sandbox, 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() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

  jpeg_decompress_struct jpeg_img;

  jpeg_source_mgr        jpeg_input_source_mgr;


  sandbox_invoke(sandbox, 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*>

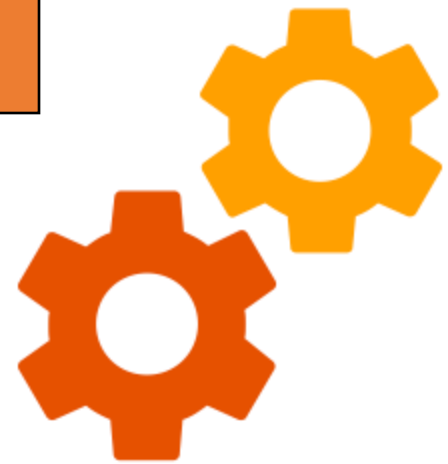
Compiles?

```
void create_jpeg_parser() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  jpeg_source_mgr        jpeg_input_source_mgr;

  sandbox_invoke(sandbox, 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() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  jpeg_source_mgr        jpeg_input_source_mgr;

  sandbox_invoke(sandbox, 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() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

  sandbox_invoke(sandbox, 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);

  }
}
```

```
void create_jpeg_parser() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

  sandbox_invoke(sandbox, 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() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

  sandbox_invoke(sandbox, 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() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

  sandbox_invoke(sandbox, 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(sandbox, jpeg_read_header, p_jpeg_img /* ... */);

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


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

    tainted_val<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() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

  sandbox_invoke(sandbox, 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(sandbox, jpeg_read_header, p_jpeg_img /* ... */);

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


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

    tainted_val<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() {
  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();
  tainted_val<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();
  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

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

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

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

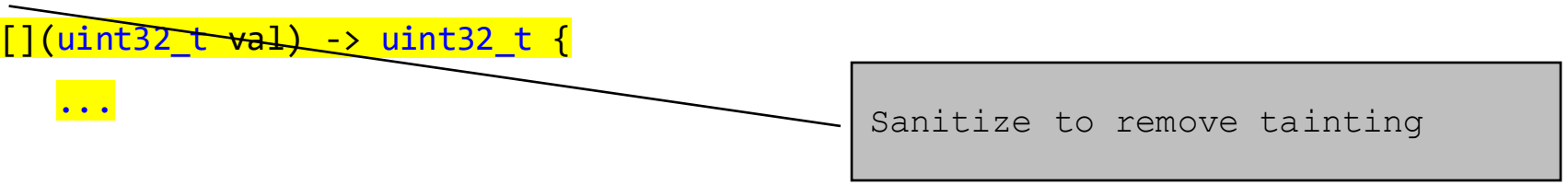
Need to remove tainting

```
void create_jpeg_parser() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

  sandbox_invoke(sandbox, 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(sandbox, 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).UNSAFE_unverified();


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

  }

}
```

Compiles?

Temporarily remove tainting

```
void create_jpeg_parser() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

  sandbox_invoke(sandbox, 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(sandbox, 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() {

  rlbox_sandbox<rlbox_noop_sandbox> sandbox;
  sandbox.create_sandbox();

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

  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

  sandbox_invoke(sandbox, 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(sandbox, 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?

```
void create_jpeg_parser() {
  rlbox_sandbox<rlbox_wasm2c_sandbox> sandbox;
  sandbox.create_sandbox();
  tainted_val<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();
  tainted_val<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

  sandbox_invoke(sandbox, 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(sandbox, 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?

This tutorial

1. Brief introduction to RLBox (20 mins)
2. Sandboxing a toy library (50 mins)
3. Preview of sandboxing in Firefox (10 mins)

Sandboxing a toy library

Get the tutorial code

```
git clone -b secdev2021 https://github.com/PLSysSec/simple_library_example
```

Works on Linux, Mac and Windows

- Requirements: CMake, C++ compiler with support for C++17

Documentation: https://docs.rlbox.dev

RLBox standard library

C/C++ standard library	RLBox standard library
`memcpy(void*, void*, size_t)`	`memcpy(rlbox_sandbox&, tainted_val<void*>, void*, size_t)`
`memset(void*, int, size_t)`	`memset(rlbox_sandbox&, tainted_val<void*>, int, size_t)`
`reinterpret_cast<type>(expr)`	`sandbox_reinterpret_cast<type>(tainted_expr)`

Unsafely add/remove tainting

Convert from type `tainted<T, T_Sbx>` to type `T`

Unsafely – Incremental porting	Safely – after porting
`tainted_val.UNSAFE_unverified()`	`tainted_val.copy_and_verify(verifier)`

Convert from type `T` to type `tainted<T, T_Sbx>`

Unsafely – Incremental porting	Safely – after porting
`sandbox.UNSAFE_accept_pointer(ptr)`	`Allowed, automatic for primitive types`

Handling callbacks – part 1 (register)

Original code	New Code

`void cb_func() {}`
 	
`lib_func(cb_func);`
	`void cb_func(int a) {}`
	
	`auto reg = sandbox.register_callback(cb_func);`
	
	`sandbox_invoke(sandbox, lib_func, reg);`
	
	`...`
	
	`reg.unregister();`

Handling callbacks – part 2 (taint params)

Original code	New Code

`void cb_func() {}`
 	
`lib_func(cb_func);`
	`void cb_func(tainted_val<int> a) {}`
	
	`auto reg = sandbox.register_callback(cb_func);`
	
	`sandbox_invoke(sandbox, lib_func, reg);`
	
	`...`
	
	`reg.unregister();`

Handling callbacks – part 3 (sandbox param)

Original code	New Code

`void cb_func() {}`
 	
`lib_func(cb_func);`
	`void cb_func(rlbox_sandbox<T_Sbx>& sandbox,`
	`    tainted_val<int> a) {}`
	
	`auto reg = sandbox.register_callback(cb_func);`
	
	`sandbox_invoke(sandbox, lib_func, reg);`
	
	`...`
	
	`reg.unregister();`

This tutorial

1. Brief introduction to RLBox (20 mins)
2. Sandboxing a toy library (50 mins)
3. Preview of sandboxing in Firefox (10 mins)

Backup

Managing structs

Library code	RLBox configuration

`struct Foo {`
`  int a;`
`  int b;`
`};`
 	
`struct Bar {`
`  int a;`
`};`
	`#define sandbox_fields_reflection_example_class_Foo(f, g, ...) \`
	`  f(int, a, FIELD_NORMAL, ##__VA_ARGS__) g() \`
	`  f(int, b, FIELD_NORMAL, ##__VA_ARGS__) g()`
	
	`#define sandbox_fields_reflection_example_class_Bar(f, g, ...) \`
	`  f(int, a, FIELD_NORMAL, ##__VA_ARGS__) g()`
	
	`#define sandbox_fields_reflection_example_allClasses(f, ...)  \`
	`  f(Foo, example, ##__VA_ARGS__) \`
	`  f(Bar, example, ##__VA_ARGS__)`
	
	`rlbox_load_structs_from_library(example);`