Static Analysis of C++ Virtual Tables (from GCC)

James Rowley, Marcus Engineering, LLC

Hardwear.io USA 2023

Step One – Get Set Up

Marcus Engineering LLC

- While I'm introducing the workshop…
- Download Ghidra:
  - https://ghidra-sre.org/

- Download the workshop files:
  - https://github.com/pixelfelon/GCCVTSRE_ghidraDemo

Welcome!

- About a year ago, my team was working on a software reverse engineering project.
  - ARM/Linux embedded system.
  - Trying to suss out how a certain digitally-tagged item was being tracked.

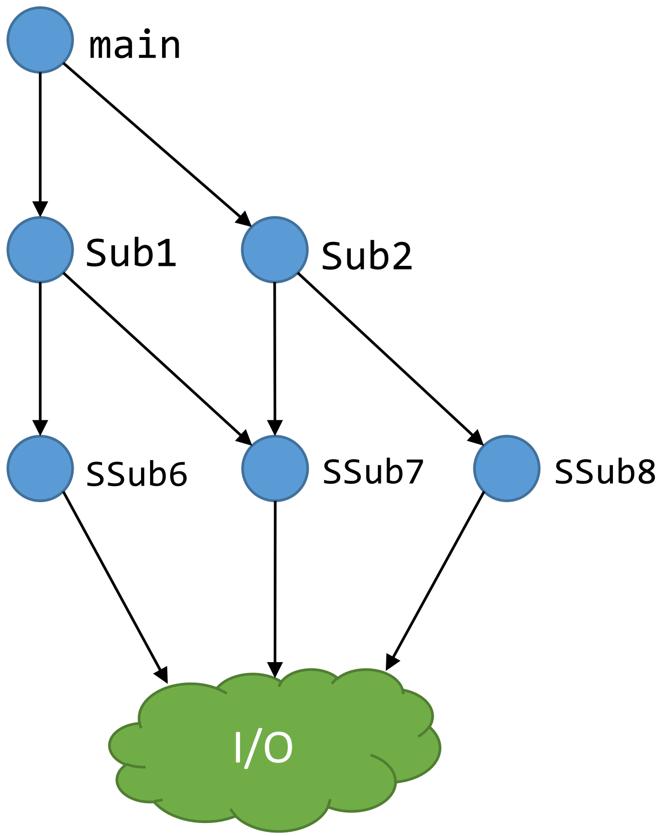
- We got the firmware out of the control console, and dug in in Ghidra…

Stuck on Indirection…

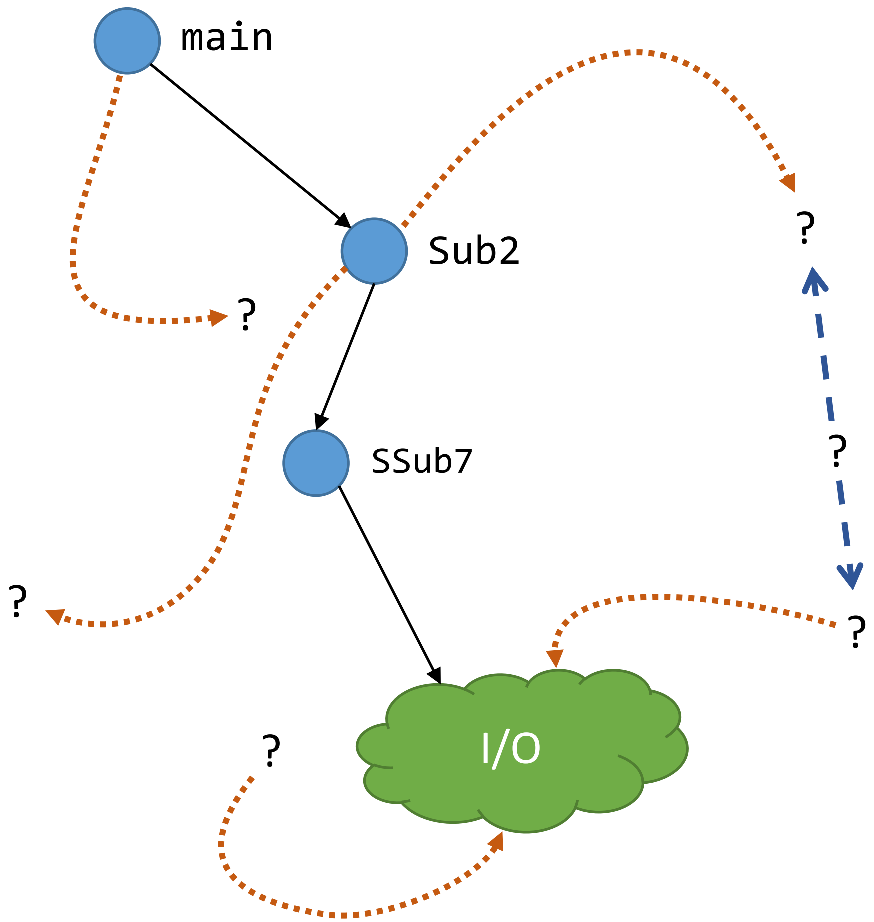
- Then we saw a lot of these jumps to computed addresses: C++ virtual calls.
  - Function calls, but we didn't know where the functions were.
- Tried to get C++-specific decompilation tools to work, and just couldn't.
  - Looked into plugins for both Ghidra and IDA Pro.

Virtual vs. Direct Calls

- Direct



- Virtual



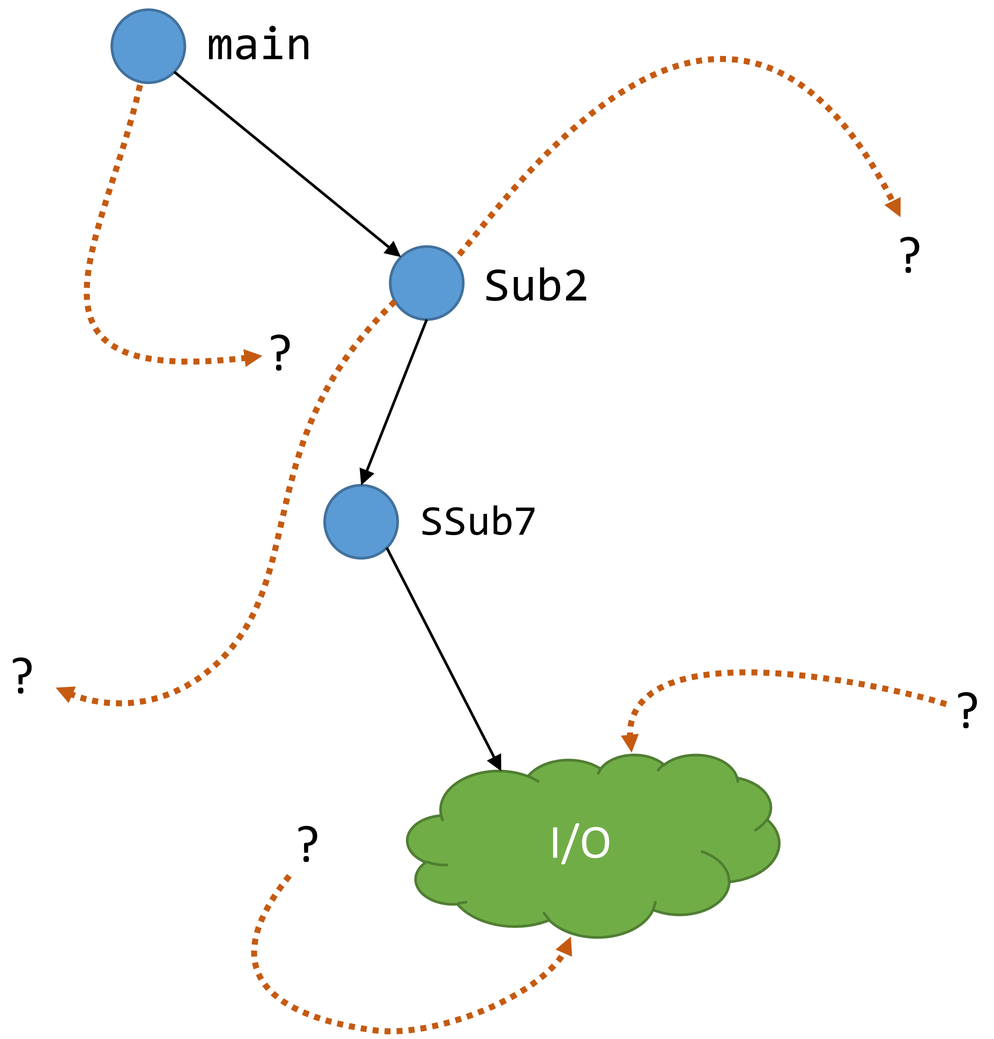
Better Try Something Else…

- Believed that it was <u>impractical</u> to *manually* analyze virtual calls and related mechanisms.
  - Hence remaining focused on C++ tools.
  - Or, doing live debugging to see the call stack.
  - But after weeks of no progress…?

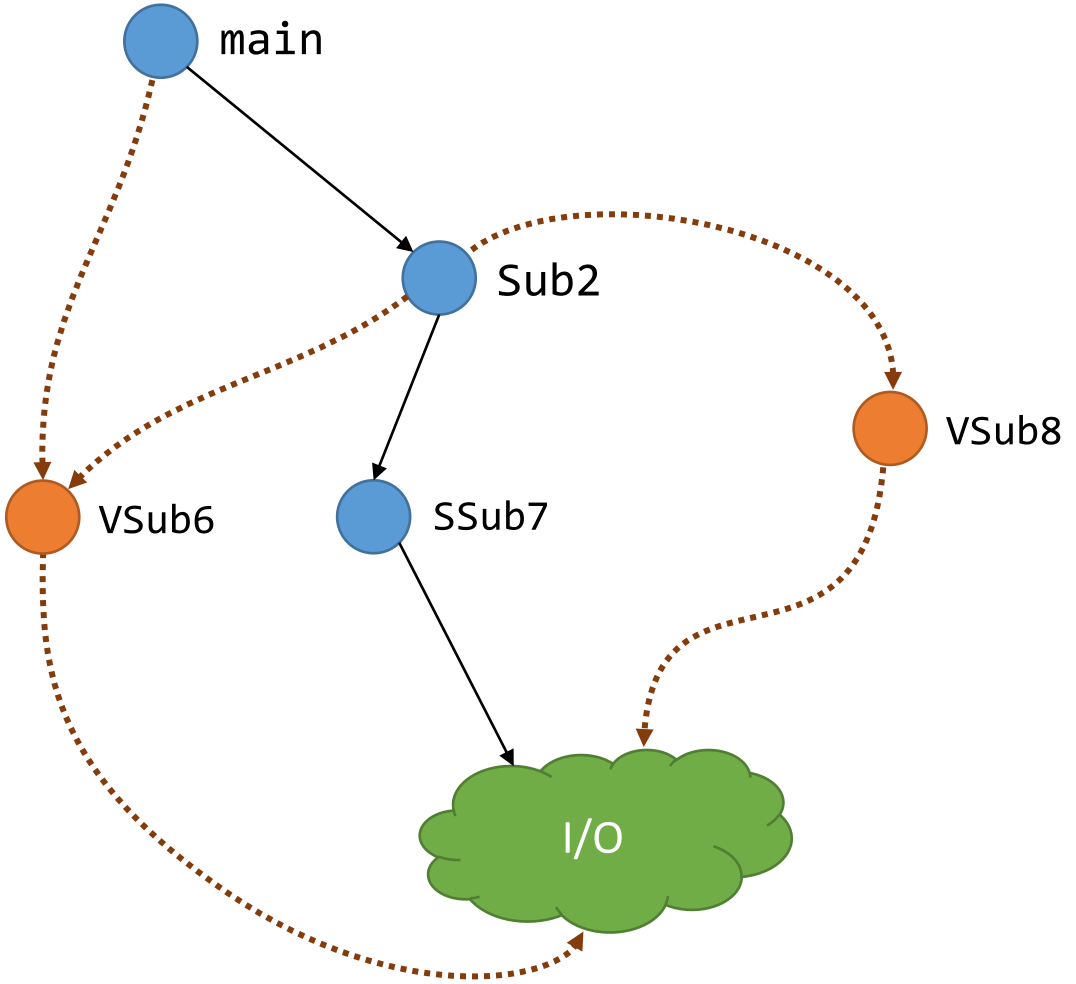
- I pushed forward on manual analysis – turns out, it's actually <u>very practical</u>.

WHAT WE'RE DOING TODAY

Marcus Engineering LLC



What we're doing today



What You Should Know

- C, especially pointers.
  - And how C may be translated to machine code.
- A basic understanding of object-oriented programming.
  - Knowing C++ would help.
- Basic Ghidra usage.

Agenda

- Introductory Example
- Caveats, etc.
- Virtual Table Primer
- Structure of Primary VTables
- Structure of Secondary VTables
- Typeinfo and Hierarchy
- Miscellanea
- Exercise *(time permitting)*

Decompiling some code…

Marcus Engineering LLC

- One day, you're decompiling some code in Ghidra.
- You see this:

```
FUN_01234567(param_1);
```

  - Ok, easy, it's calling some function at 0x01234567.

- But you also see this:

```
(**(code **)(*param_1 + 0x8))(param_1);
```

  - What's it actually calling..?

So what is it?

- This is the decompilation of a binary originally written in C++. You're looking at calls to what were originally methods on a C++ class.
  - And as it so happens, that class has virtual methods.
- With an empty structural type for `this`, Ghidra will decompile a virtual call like so:

```
(**(code **)(*(int *)this + 0x8))(this);
```

- What's getting called???
  - The function pointer at the address stored in "this" plus eight...?
  - Yep, it's a virtual function.

EXAMPLE – Annotation I

Marcus Engineering LLC

- Original decompilation:

```
(**(code **)(*(int *)this + 0x8))(this);
```

- Now define a type for `this`:

```
typedef struct {
    void ** vtable;
} Base;
```

- New decompilation:

```
(*(code *)(this->vtable[1]))(this);
```

EXAMPLE – Annotation II

Marcus Engineering LLC

```
(*(code *)(this->vtable[1]))(this);
```

- Now define a type for `this->vtable`:

```
typedef struct {
  code * foo;
  code * bar;
} Base::vtable-funcs;
```

```
typedef struct {
  Base::vtable-funcs
      * vtable;
} Base;
```

- Final decompilation:

`(*this->vtable->bar)(this);` – Nice!

C++ in terms of C

- `(*this->vtable->bar)(this);` is a lot easier to understand than the original was…
  - But it's probably not how the original code looked.
  - More like… `bar();`.
- But Ghidra decompiles C, not C++.
  - As is the case with most decompilation tools.
  - So, we need to reimagine all of C++'s features in terms of pure C.
  - Which is actually pretty easy! Just very verbose.

Prefacing Miscellanea

The important odds and ends!

How to identify a C++ binary?

- It’s all about mangled names.
  - `_ZN3Foo3barEv` or something like that.
  - Check the ABI, very intricate scheme.
- Will have lots of linker symbols exhibiting this sort of mangling.
  - If there are no linker symbols, (fewer of) these names can still be found as const strings.
  - If on a non-GCC platform, the mangling may look very different, but should still be present.

Some Caveats

- These techniques were originally developed on 32-bit ARM binaries compiled with GCC 4.8/4.9.
  - They seem to be generally applicable to other versions and platforms of GCC.
  - Indeed, our exercise today will be on x86_64.
- Ghidra seems to be better at picking up on objects and vcalls on x86 than on ARM.
  - So, the initial decompilation of an x86 binary may be different and more complete than shown here.

More Caveats

Marcus Engineering LLC

- GCC uses the Itanium C++ ABI.
  - The Itanium ABI is **not** universal on x86.
    - That's why this workshop is about GCC.
  - MSVC binaries could be *completely different.*
    - I haven't checked.
  - But, Itanium ABI is more common on other platforms.
    - It's the official standard for ARM.

- Also, I've never really developed in C++…
  - But I have a lot of experience in C, and OOP in Python.
  - So, I learned large portions of the C++ language from the ABI and decompiled binaries.

Hit the books!

- The Itanium C++ ABI Specification is an *invaluable* resource for working with vtables emitted by GCC.
  - Particularly Section 2.5, “Virtual Table Layout”.
- https://itanium-cxx-abi.github.io/cxx-abi/abi.html
  - This presentation cannot and will not supplant it!

- Yes, that’s the *Itanium* C++ ABI. It is widely used, even though nobody uses Itanium anymore.
  - The ARM ABI and GNU GCC both specifically call it out.
  - Though, GCC extends it a bit… good luck there!

We’re not covering the whole ABI!

- We will ***not*** be discussing classes with virtual bases.
  - They complicate static analysis.
  - They don’t seem to be very common.
    - *We have actually dealt with a few now, it’s not that bad.*
  - See Category 3/4 vtables in Section 2.5.3 of the ABI.
- So, the vtables will be fairly simple, and we’ll never deal with construction vtables or VTTs.

Key Terms

- Object
- Class
- Concrete Type
  - *Most-Derived Class*
- Virtual Method
- Pure Virtual Method
- Thunk
- Emitted
  - *Binary Code/Data*
- Virtual Base
- Ghidra

Key Terms – Quick Glossary

- Object Data – the data actually stored in memory for an instance of an object.
  - i.e., all non-static fields.
  - Representable as a C struct.
- Subobject – a section of object data belonging to a particular class in the object's type hierarchy.

Key Terms – Glossary

- Object – an instance of a class.
- Class – the type of an object.
- Most-Derived Class – when considering a specific *object's* class hierarchy, the single class which is not a base of any other class. Its "type", more or less.
- Object Data – the data actually stored in memory for an instance of an object.
  - i.e., all non-static fields.
  - Representable as a C struct.
- Subobject – a section of object data belonging to a particular class in the object's type hierarchy.
- Virtual Method – a method on a class, which can be overridden in a subclass.
  - i.e. what code is called depends on the object type.
  - Can be overridden (non-virtual methods cannot be).
- Thunk – a very small function which has the sole purpose of calling another function. One might also call it a "shim".

Key Terms – Glossary

- Emitted – actually turned into machine code or data by the compiler.
- Pure Virtual Method – a virtual method which does not have an implementation in its containing class.
  - Calling it would be a fatal error (fine to call an override, of course).
- Virtual Base – a base whose subobject will exist exactly once in the most-derived class, regardless of how many times it appears in the hierarchy.
  - We're not going to deal with these!
- Typeinfo Structure – some static, constant data emitted by the compiler which describes a type (usually a class).
  - Describes a type sufficiently for comparing it to other types…
  - But not sufficiently for full runtime reflection (darn!).
- Ghidra – software reverse-engineering framework with disassembler and decompiler.
  - It's our tool for this workshop.

The Basics

What’s a VTable?

Why have "VTables"?

Marcus Engineering LLC

- C++ allows for "virtual" functions that can be **overridden** in subclasses, changing behavior.
  - And objects of a derived type can be treated as if they were objects of the base type.
- VTables are the fundamental mechanism that allows subtype polymorphism in C++ (in GCC).
- So at runtime, somehow, `obj->bar();` needs to call `Base::bar` or `Derived::bar` depending solely on the type of `obj`.
  - This is what `obj`'s VTable accomplishes.

What *is* a "VTable"?

- Virtual Table – an array of function pointers to the <u>implementations of all virtual methods</u> in a class.
  - e.g., base methods, method overrides, concrete implementations of pure virtual methods.
  - Also, contains information about the layout of subobjects, and type hierarchy.
  - **Constant**, emitted by the compiler; used at runtime.

When will you see a VTable?

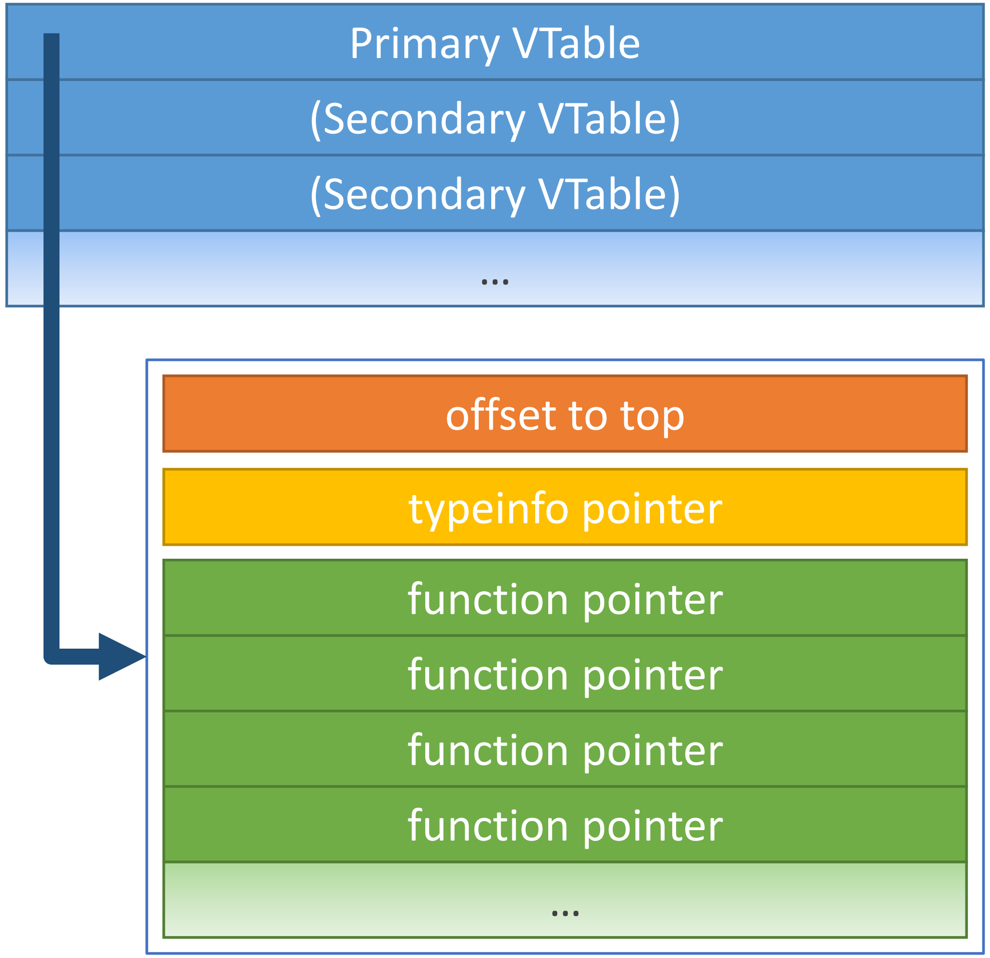
- Not all classes have a vtable.
- To have a vtable, the class must:
  - Declare a virtual function, or
  - Inherit a virtual function.
- Doesn't matter if bases are declared virtual or not; if a base has a virtual function:
  - It has a vtable.
  - Its inheriting class will inherit that virtual function.
    - It may or may not override it.
  - Its inheriting class **will** have a vtable.

Basic VTables

(Classes with at most one non-virtual base)

What Do VTables Look Like?

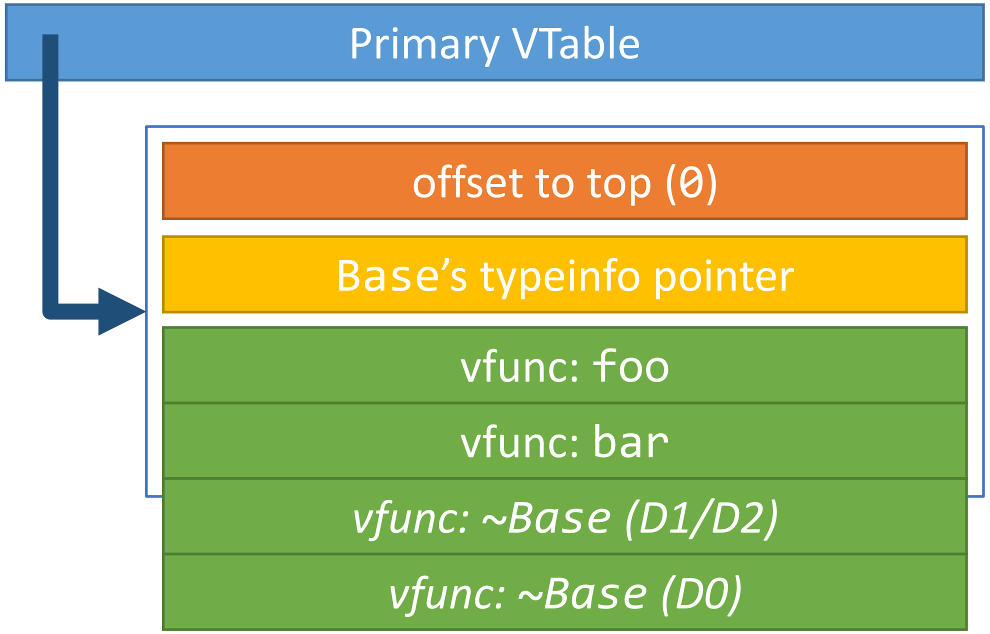
- For now, let's just focus on the Primary VTable:
- "Offset to top" – zero.
- Typeinfo pointer.
  - To compiler emitted typeinfo structure for the class.
- Function pointers.
  - To methods which will accept object data from exactly this class as their `this` parameter.
- For x86_64, pointers are on an 8-byte alignment.



Primary VTables – Layout

- Every class with virtual functions has one.
- Virtual functions appear in source order.
  - Virtual functions of the primary base classes appear first, in their original order.
  - Virtual destructors get two entries – the base- and complete-object destructor, in that order.

```
class Base {
        virtual void foo ();
        virtual void bar ();
        unsigned int b;
        virtual ~Base() {}
}
```



Typing VTables as C Structures

- VTables can have **structure type** annotations applied in Ghidra.
  - Once you've taken the time to make them, they propagate to <u>everywhere that class is used</u>, and provide more meaningful decompilation.

- Only bother with the function pointer array.
  - Nothing really references the RTTI before it.

Base's VTable as a C Structure

Marcus Engineering LLC

```
class Base {
    virtual void foo ();
    virtual void bar ();
    unsigned int b;
}
```

- The VTable will have just foo and bar.

```
typedef struct {
    void (*foo)(Base * this);
    void (*bar)(Base * this);
} Base::vtable-funcs;
```

Demo in Ghidra

The Not-So-Basics

Secondary VTables

Secondary VTables – When?

Marcus Engineering LLC

- The derived class will have a vtable for *each* base class with virtual functions.
- If there's multiple such bases, there's a secondary vtable.
  - In the example to the right, "Base-in-Derived" is the official name for such vtable.

```
class Base {
    virtual void foo (void);
    virtual void bar (void);
    unsigned int b;
}

class Quirk {
    virtual void quirk (void);
    void * Q;
}

class Derived : Quirk, Base {
    virtual void baz (void);
    void bar (void);
    unsigned char d;
}
```

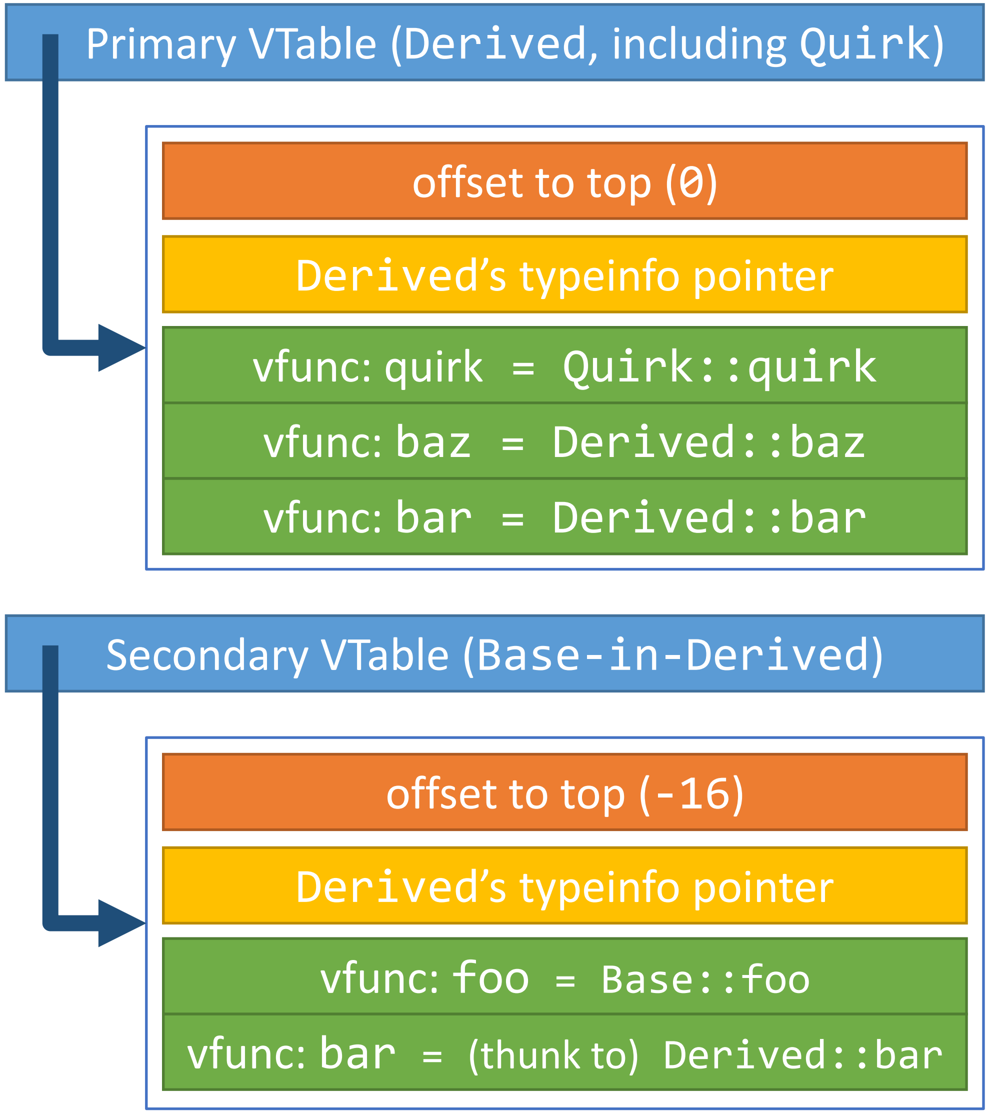
Primary Vtable – Derived (and Quirk)
Secondary Vtable – Base-in-Derived

Secondary VTables – What?

```
class Base {
        virtual void foo (void);
        virtual void bar (void);
        unsigned int b;
}

class Quirk {
        virtual void quirk (void);
        void * Q;
}

class Derived : Quirk, Base {
        virtual void baz (void);
        void bar (void);
        unsigned char d;
}
```



Secondary Vtables – Why?

```
class Base {
        virtual void foo (void);
        virtual void bar (void);
        unsigned int b;
}

class Quirk {
        virtual void quirk (void);
        void * Q;
}

class Derived : Quirk, Base {
        virtual void baz (void);
        void bar (void);
        unsigned char d;
}
```

Quirk Object Data Layout

Quirk *	vtable * vtable
	void * Q

Base Object Data Layout

Base *	vtable * vtable
	unsigned int b

Derived Object Data Layout

Quirk *, Derived *	vtable * vtable
	void * Q
Base *	vtable * vtable
	unsigned int b
	unsigned char d

```
Derived obj;
assert((void *)dynamic_cast<Derived *>(&obj)
    == (void *)dynamic_cast<Base *>(&obj));
// Would fail!
```

Secondary Vtables – Why?

Marcus Engineering LLC

- It’s all about the layout of the object data.
  - New fields go last, but…
  - Only one base subobject can go first.
- Need some kind of adjustment to `Derived` if we pass it to something expecting a `Base`.
  - Virtual functions are still overridden, though.
  - So that adjustment has to be undone.

Secondary Vtables – Why?

- The `this` pointer needs adjustment between `Derived*` and `Base*`.
  - Consumers of `Base*` need a `Base*`, not `Derived*`.
  - The implementations of `Derived`'s methods need a `Derived*`, even if the caller has it as a `Base*`.

- The secondary vtable makes it all work.
  - It can point to special code to handle this…

Secondary Vtables – How?

- When converting to `Base*`, you get a pointer to the `Base` subobject, with its <u>secondary vtable</u>.

- The secondary vtable contains pointers to **thunks**, instead of the actual methods on `Derived`.
  - These thunks accept a `Base*`, <u>convert it</u> back to a `Derived*`, and call the associated method on `Derived`.

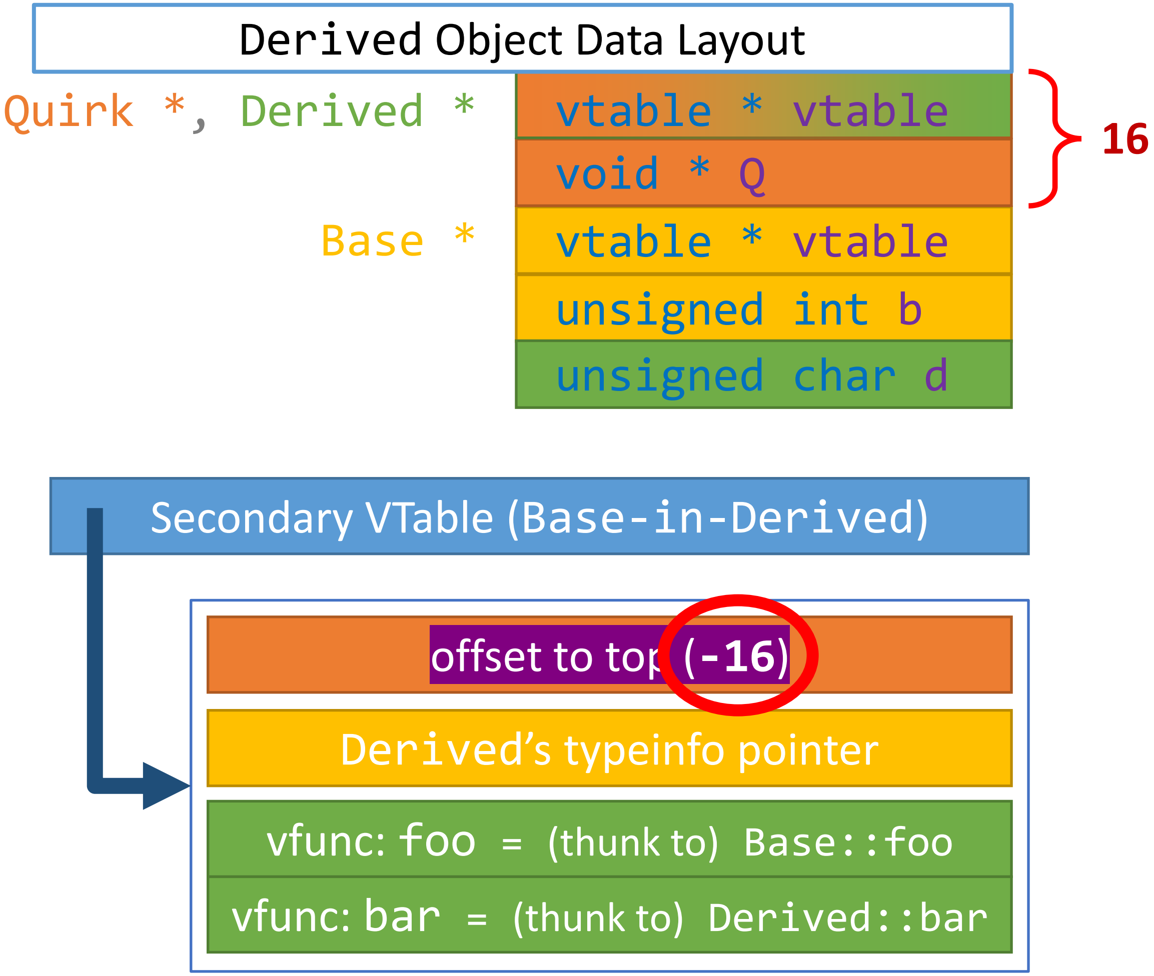
Notice The Offset

Marcus Engineering LLC

```
class Base {
        virtual void foo (void);
        virtual void bar (void);
        unsigned int b;
}

class Quirk {
        virtual void quirk (void);
        void * Q;
}

class Derived : Quirk, Base {
        virtual void baz (void);
        void bar (void);
        unsigned char d;
}
```



Secondary Vtables – Why, really?

Marcus Engineering LLC

- The secondary vtable contains pointers to **thunks**, instead of the actual methods.
  - Consumers don't need to adjust the `this` pointer at all, because the thunks will.
- So, consumers <u>don't need to consider</u> an object's concrete type for overriding to work.

Quite Common…

- In the applications we were reverse engineering, probably half of the classes we encountered had secondary vtables.

- They really liked to use Qt for everything, but not base interface classes.
  - So you inherit from QObject, and then the interface, each with virtual functions.

Demo in Ghidra

Keep Track of Types!

- You've identified a `Derived* obj`.
  - In the binary, it may at *any moment* get upcasted into a `Base*`!
  - Their layout is *not compatible*, nor is the layout of their vtables.

- So, if `obj` gets manipulated, and then a vcall happens… make sure you know what type it is *right then*, so you know which vtable it's using.
  - Sometimes Ghidra won't recognize that an operation has changed the type of a variable. So you can't change the before/after type separately. In this case, just use lots of comments.

Type Hierarchy

(and how to figure it out)

Typeinfo Structures

- Constant “RunTime Type Information” emitted by the compiler for every class.
  - Required by the ABI.

- Contains links to its base classes.
- Contains the <u>name of the class</u>.
  - That’s really helpful in a stripped binary!

Typeinfo Structures

- Every vtable has a pointer to its class's typeinfo.
  - So, you can propagate the name from the typeinfo.
- Two particularly helpful varieties:
  - `__si_class_type_info` – for **single base**.
  - `__vmi_class_type_info` – for **multiple bases**.
- Those link to the typeinfo of the base classes.
  - And of course, they name the class.

Typeinfo Structures - Reference

- Simple C layouts of C++ ABI class typeinfos:

```
typedef struct {
        void ** vtable;
        const char * name;
        typeinfo * base_type;
} __si_class_type_info;

typedef struct {
        void ** vtable;
        const char * name;
} __class_type_info;

typedef struct {
        void ** vtable;
        const char * name;
        vmi_flags flags;
        uint32_t base_count;
        base_class_type_info[] base_info;
} __vmi_class_type_info;
```

```
typedef struct {
        bool non_diamond_repeat:1;
        bool diamond_shaped:1;
        int  :2;
        bool flags_unknown:1;
        int  :27;
} vmi_flags;

typedef struct {
        __class_type_info * base_type;
        offset_flags offset_flags;
} base_class_type_info;

typedef struct {
        bool virtual:1;
        bool public:1;
        int offset:30;
} offset_flags;
```

Discovering VTables

Marcus Engineering LLC

- Sometimes, there are no linker symbols…
- The property that each vtable has a pointer to a typeinfo, and each typeinfo has a vtable too, is *very useful*!
  - Start by finding and labelling the standard typeinfo vtables.
  - `__class_type_info::vtable-funcs`
  - `__si_class_type_info::vtable-funcs`
  - `__vmi_class_type_info::vtable-funcs`
- Important: put a label at the <u>start of the function pointers</u>, since that's what typeinfo objects will point to.
- Now references to these typeinfos will be clearly visible.

What VTables and Typeinfo Look Like

Typeinfo

- Absolute pointer
  - (To typeinfo's vtable)
- Absolute pointer
  - (To type name string)
- Maybe more pointers
  - (To parent typeinfos)

VTable (primary)

- Zero
- Absolute pointer
  - (To typeinfo)
- One or more absolute pointers
  - (To virtual functions)

Proximity in the Binary

- Const data coming from a single translation unit is usually all close together.
  - Including vtables and typeinfos.
  - Same goes for program text?
- So if you find something interesting, the nearby data is probably related.

Proximity is key.

- Part of what we had to analyze was a huge binary with no linker symbols.

- Being able to recognize that some things were related because they were nearby was super helpful – it multiplies what you learn.

Naming the Const Data

- Once you’ve found the typeinfo, the class name, and the vtable, you should label it.
- I like to use these names:
  - `<class>::typeinfo`, `::typeinfo-name`
  - `<class>::vtable`, `::vtable-funcs`
- Now everywhere those are used, you have a nice descriptive name.

Working Up the Chain

- If you've got a class with some pure virtual methods, you can't tell what they do.
- But you can use the typeinfo to look for a subclass that implements them…

- Also just generally good to annotate vtables up and down the inheritance tree.

Demo in Ghidra

Marcus Engineering LLC

GHIDRA

Demo Recap

- We found mangled names as const strings.
  - Like “4Base”.
- These names were used in typeinfo structures.
- The typeinfo structures were used in vtables.
- And finally, the vtables were used in constructors.

Miscellanea

(subtitle)

When `this` isn't first.

- If the return type is *non-trivial*, the `this` pointer may be preceded by a `RETURN` pointer.
  - Also for constructors with virtual bases.

- Also, double words – check your ABI. Ghidra may well get it wrong; it certainly does on ARM.

Template Classes in Ghidra

- Instances of template classes will frequently have mutually-compatible object data.
  - It may even be guaranteed by the definition of the class.
- It’s tempting to just make one struct in Ghidra, and typedef the instantiations to it!
- This will break the decompiler!
  - It cannot seem to handle “this” (specifically from __thiscall) pointing to anything other than a struct.
  - Worse, it can’t handle that scenario anywhere in the call tree…
- <u>Instead, I suggest:</u>
  - Making the one struct with the concrete object layout.
  - Keeping all the template instantiation object data structs.
  - Adding to each such struct, the layout struct as its sole member.

GCC Extensions to Itanium C++ ABI

- Sometimes you'll see a mangling that just *does not make sense,* according to the "official" ABI.
  - Of course, it's hardly official, it's just a community-maintained GitHub repo.

- Known extensions:
  - L at the start of a function mangling:
    - Indicates a static function.
    - e.g.: "`_ZL3foov`" → `static void foo (void);`
  - C4 as a constructor name:
    - Indicates a "base-object allocating constructor".
    - e.g.: "`_ZLN3FooC4Ev`" → `class Foo : Base { __? Foo () {} }`
    - Well, the C++ half of that is notional. But you get the idea.

Itanium C++ ABI – Available in PDF!

Marcus Engineering LLC

- I’ve typeset the ABI, which in its native form is one big webpage.
  - https://github.com/itanium-cxx-abi/cxx-abi/files/8994612/Itanium.CXX.ABI.June2022.pdf

- Easier to print, easier to bookmark.

Activity

SRE Challenge

Final Demo/Activity

- A little CLI “hashing” program.
- Enter some text, get a number.

- What algorithm is it using?

- Stripped of linker symbols.
  - But there are library imports.

Just to make it interesting…

Marcus Engineering LLC

- The algorithm is non-standard.
  - Won’t have any luck googling the constants…

- ~~I’ve had a colleague randomize some details, so~~ this isn’t totally rehearsed.
  - It’s been a couple weeks and I don’t remember what I wrote. Close enough!

Final Demo/Activity in Ghidra

Thank you for coming!