Reversing C++ Decompiled Code

Author: ThreatBlogger

INTRODUCTION

This report will demonstrate the process of reverse engineering object-oriented programming concepts such as Classes, Inheritance and Virtual functions. Using a simple piece of C++ code I will reconstruct the decompiled code to an understandable compliable format using tools such as the Snowman Decompiler and IDA PRO.

The Main Method

- When the file is decompiled in the Snowman Decompiler we can go straight to the main method as shown.
- The code may not be understandable to the untrained eye.
- The classes are converted to functions. We can see here two functions and two created objects. The objects are converted to function pointers and executed.

```
/* .text */
int64_t text(void** rcx, void** rdx, void** r8, void** r9) {
    int64_t v5;
    void** rax6;
    int64_t v7;
    void** rax8;
    void** rax9;
    void** rax10;

    __main(rcx, rdx, r8);
    rax6 = text__Znwy(8, rdx, r8, r9, v5);
    *reinterpret_cast<void***>(rax6) = reinterpret_cast<void**>(0);
    _ZN1BC1Ev(rax6);
    rax8 = text__Znwy(8, rdx, r8, r9, v7);
    *reinterpret_cast<void***>(rax8) = reinterpret_cast<void**>(0);
    _ZN1CC1Ev(rax8);
    rax9 = *reinterpret_cast<void***>(*reinterpret_cast<void***>(rax6));
    rax9(rax6);
    rax10 = *reinterpret_cast<void***>(*reinterpret_cast<void***>(rax8));
    rax10(rax8);
    system("pause");
    return 0;
}
```

The Classes

- Having a quick look at functions related to main, which would have been class methods, we can see three functions, their names are A(), B() and C() .
- I rename these functions as _A, _B and _C.

```
/* A::A() */
void _ZN1AC2Ev(void** rcx);

/* B::B() */
void _B(void** rcx) {
    _ZN1AC2Ev(rcx);
    *reinterpret_cast<void***>(rcx) = reinterpret_cast<void**>(0x490510);
    return;
}

/* C::C() */
void _C(void** rcx) {
    _ZN1AC2Ev(rcx);
    *reinterpret_cast<void***>(rcx) = reinterpret_cast<void**>(0x490530);
    return;
}
```

INHERITANCE

Looking at the code, I can see Inheritance is implemented which is an object-oriented programming concept in which parent-child relationships are established between classes. Child classes inherit function and data from the parent base class.

We have three classes here, _A is the base type and both _B and _C are subtypes which inherit the characteristics and behaviours of _A.

All three classes have been decompiled as type void, we change _A to type **struct**, as structs are the same as classes in C++, without changing the whole structure of the code. A quick tidy up of the classes makes things more clear:

- There are different methods to achieve Inheritance the method I have chosen here is include a struct pointer as an argument in the _B and _C functions.
- From experience I see a pointer to two Virtual functions in class _B and _C.

```
/* A::A() */
struct _A {      };

/* B::B() */
void _B(void** rcx) {
   _A(rcx); //super
   *reinterpret_cast<void***>(rcx) = reinterpret_cast<void**>(0x490510); //virtual function
   return rcx;
}

/* C::C() */
void _C(void** rcx) {
   _A(rcx); //super
   *reinterpret_cast<void***>(rcx) = reinterpret_cast<void**>(0x490530); //virtual function
   return rcx;
}
```

VIRTUAL FUNCTIONS

Using Virtual functions is how Polymorphism is implemented. A standard function is normally executed at compile time. A Virtual function is one that can be overridden by a subclass and whose execution is determined at runtime. This is used to simplify complex programming tasks. A Virtual function has to be named “Virtual” in the C++ code.

Virtual Function Table

The C++ compiler adds special data structures when it compiles code to support virtual functions, these data structures can be called *virtual functions tables* of *vtables* or *vftables.* A virtual table is a lookup table of functions pointers.

Virtual Functions In IDA PRO

Looking in IDA PRO at class B we see its calling a virtual function in class A.

```
; Attributes: bp-based frame

; __int64 __fastcall B(B *__hidden this)
public _B
_B proc near

arg_0= qword ptr  10h

push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+arg_0], rcx
mov     rax, [rbp+arg_0]
mov     rcx, rax        ; this
call    _A
mov     rax, [rbp+arg_0]
lea     rdx, off_490510 ; pointer to virtual function
mov     [rax], rdx
add     rsp, 20h
pop     rbp
retn
_B endp
```

Double clicking on **off_49510** we see the following:

```
                db    0
                db    0
                db    0
                db    0
off_490510      dq offset _ZN1B5printEv ; DATA XREF: _B+1C↑o
                                        ; B::print(void)
```

The address **off_49510** is a virtual table and it stores a pointer to a virtual function called print. It looks to be a virtual print function, if we double click on **_ZN1B5printEv** we see the following:

```
; __int64 __fastcall print(B *__hidden this)
public print
print proc near

this= qword ptr  10h

push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+this], rcx
lea     rdx, aClassB    ; "Class B"
mov     rcx, cs:_cout   ; std::ostream *
call    cout            ; << "Class B" ;
mov     rdx, cs:endl
```

This print function translates to:

```
void print(){  std::cout << "class B" << endl;   }
```

Reconstructing Classes

With the information we now have we have a better idea of how this program is constructed. Going back to the classes I changed the following:

Class _b and _C inherits _A object pointer through a function argument and also returns a struct object. I renamed the two virtual function addresses to vftable1 and 2 so they can be used in this program.

I introduced a function pointer to Struct _A called "print", this will be used later in main as a pointer to either vftable 1 or 2.

```
/* A::A() */
struct _A {

 void (*print)();

  };

/* B::B() */
struct A* _B(struct A* rcx) {

    *reinterpret_cast<void***>(rcx) = reinterpret_cast<void**>(vftable1); //virtual function
    return rcx;
}

/* C::C() */
struct A* _C(struct A* rcx) {

    *reinterpret_cast<void***>(rcx) = reinterpret_cast<void**>(vftable2); //virtual function
    return rcx;
}
```

I will now create these functions from the information I gathered in Ida Pro:

```
/* A::A() */
struct _A {  void(*print)();   };


void vftable1() {   std::cout << "Class B" << std:endl;    }
void vftable2() {   std::cout << "Class C" << std:endl;    }

/* B::B() */
struct _A* _B(struct _A* rcx) {

    *reinterpret_cast<struct _A**>(&rcx->print) = reinterpret_cast<struct _A*>(vftable1);  //virtual functon
    return rcx;
}

/* C::C() */
struct _A* _C(struct _A* rcx) {

    *reinterpret_cast<struct _A**>(&rcx->print) = reinterpret_cast<struct _A*>(vftable2);  //virtual function
    return rcx;
}
```

The Main Method

We can now return back and construct the main function with what we know from the information we gathered from the classes. The names of functions that are decompiled are usually mangled. After a bit of a tidy up makes it a bit more understandable.

```
/* .text */
int64_t text(void** rcx, void** rdx, void** r8, void** r9) {
    int64_t v5;
    void** rax6;
    int64_t v7;
    void** rax8;
    void** rax9;
    void** rax10;

    __main(rcx, rdx, r8);
    rax6 = malloc(8); //memory allocation
    *reinterpret_cast<void***>(rax6) = reinterpret_cast<void**>(0);//object =0
    _B(rax6);
    rax8 = malloc(8);//memory allocation
    *reinterpret_cast<void***>(rax8) = reinterpret_cast<void**>(0); //object =0
    _C(rax8);
    rax9 = *reinterpret_cast<void***>(*reinterpret_cast<void***>(rax6)); //ptr ot B class
    rax9(rax6); // executes -> function
    rax10 = *reinterpret_cast<void***>(*reinterpret_cast<void***>(rax8)); //ptr to C class
    rax10(rax8);// executes -> function
    system("pause");
    return 0;
}
```

At the start of main, we see memory allocation is taking place, two objects are created and passed to two functions. The address of those objects are used in a function pointer, here is an example of a function pointer:

```
typedef int func(void);  //create function

func* f =(func*))0xabcd1234;  //get the function address

f();  //call function
```

Conclusion

Understanding object-oriented programming and how it works is the first step in reconstructing Classes, Inheritance and Virtual functions. Using tools such as IDA PRO and Snowman Decompiler helps reconstructing code and making it a lot easier to understand.

Reversed Code

Here is the full working code:

```
#include <iostream>
#include <windows.h>
#include <inttypes.h>
/* A::A() */
struct _A { void(*print)(); };
void vftable1() { std::cout << "Class B" << std::endl; }
void vftable2() { std::cout << "Class C" << std::endl; }
/* B::B() */
struct _A* _B(struct _A* rcx) {
*reinterpret_cast<struct _A**>(&rcx->print) = reinterpret_cast<struct _A*>(vftable1);//virtual functon
return rcx;
}
/* C::C() */
struct _A* _C(struct _A* rcx) {
*reinterpret_cast<struct _A**>(&rcx->print) = reinterpret_cast<struct _A*>(vftable2);//virtual function
return rcx;
}
int main() {
int64_t v5;
struct _A* rax6;
int64_t v7;
struct _A* rax8;
struct _A* rax9;
struct _A* rax10;
// __main(rcx, rdx, r8);
rax6 =(struct _A*) malloc(8);//allocates memory
*reinterpret_cast<void***>(rax6) = reinterpret_cast<void**>(0);//object=0
_B(rax6); //carries object to C funtion
rax8 = (struct _A*) malloc(8);//allocates memory
*reinterpret_cast<void***>(rax8) = reinterpret_cast<void**>(0);//object=0
_C(rax8); //carries object to C funtion
rax9= _B(rax6);//returns pointer to virtual functon
rax9->print();//executes -> function
rax10 = _C(rax8);//returns pointer to virtual functon
rax10->print();//executes -> function
free(rax6);
system("pause");
return 0;
}
```