Week 2: Reversing Basic C++

Author: Nguyen Anh Khoa

Instructor: Dang Dinh Phuong

Introduction

This week target is to get familiar with reversing, ASM language, IDA usage. The job was to try reading, and make sense of the binary produced from last week exercises. I prepared two build, debug and release (with debug information) both x86, and tried to read the binary using IDA. In debug build, Visual studio wrap function with debug call at top of the function, take a look into this function, it will call `ds:GetCurrentThreadId` , functions like this will be skipped because it is not related to our reversing.

This article assumed that readers know, at least a little, about:

- The ASM language, common operation
- C programming language
- C++ programming language with OOP
- Basic IDA usage

Finding main

It it supprising that IDA was not able to recognize our `void main(int argc, char** argv)` function, however by finding strings, we can find the reference to the string, which is in our program executation path. In these exercises, the string is located directly in main, so finding reference to string will return to our main function. If strings are in another function, find the function cross-reference and follow up to find our main function, using (List cross-reference from/to).

Calling convention

The arguments are pushed on stack (exception on float/double arguments) from last to first arguments. `void foo(int a, int b, int c);` will be push on stack by order:

```
push c
push b
push a
call foo
```

So if we know the function fingerprint, we can find the correspondent variables. Or if we don't know the function fingerprint, we can use the stack to indentify the function fingerprint. The result of the call is stored in `eax` register, look for the `mov [], eax` to see where the value of the function call is stored.

The call convention can different, from first to last could be `rcx, rdx, r8, r9, ...stack` . Reference microsoft/cpp/x64-calling-convention, on `x86_64` default using MSVC. We also have the order as `rdi, rsi, rdx, rcx, r8, r0, ...stack` . Order compiler may use different calling convention, or using a different calling convention. Reference scc.ustc.edu.cn/intel/compiler_c, microsoft/cpp/argument-passing-and-naming-conventions. From the microsoft link, we can verify that the above calling convention was `__stdcall: Pushes parameters on the stack, in reverse order (right to left)` .

The stack pointer and base pointer

This part assumed that we are using the `__stdcall` calling convention.

The stack is a crucial component, it is commonly used to store local variables, passing parameters, saving stack frame. There is only one stack for one program/process, it is a static array. In MSVC, the size is default to 1MB, when the stack is used out a `stack overflow` happens.

On entering a function, stack pointer will be reduced to save local variables with offset, when exit the function, stack pointer will be added up. Local variables can be reference by using the stack pointer and the offset.

```
+------+----------+
| 0x50 | arg_e    |
+------+----------+
| 0x4C | arg_d    |
+------+----------+
| 0x48 | ret_addr |
+------+----------+
| 0x44 | var_c    |
+------+----------+
| 0x40 | var_b    |
+------+----------+
| esp  | var_a    |
+------+----------+
```

With stack pointer is `esp` , we can calculate the address of `var_a, var_b, var_c, ret_addr, arg_d, arg_e` by offseting from `esp` . The variable where `esp` is is called the top of the stack. But we frequently see another register used for offseting from stack, the `ebp` base pointer. When using `ebp` the stack layout will be:

```
+------+----------+
| 0x54 | arg_e    |
+------+----------+
| 0x50 | arg_d    |
+------+----------+
| 0x4C | ret_addr |
+------+----------+
| ebp  | old ebp  |
+------+----------+
| 0x44 | var_c    |
+------+----------+
| 0x40 | var_b    |
+------+----------+
| esp  | var_a    |
+------+----------+
```

The use of `ebp` is to offset to variables, arguments, and return address. Using `ebp` will simplify the offset, as `var_a, var_b, var_c, ret_addr, arg_d, arg_e` offset will be the same regardless of how `esp` move when pushing, poping. The routine will be as follow:

```
main:
  ; main function
  push arg_e
  push_arg_d
  call foo      ; place ret_addr in stack
  add esp, 8
  ; main function

foo:
  push ebp      ; saving old ebp
  mov ebp, esp  ; set ebp, save old esp
  sub esp, 12
  ; foo function
  add esp, 12
  mov esp, ebp  ; retrieve old esp
  pop ebp       ; retrieve old ebp
  retn          ; jump to ret_addr
```

The `ebp` usage is less and less due to compiler improvements. I don't understand why on release build, the `ebp` is not being used anymore.

References links:

- All about ebp
- Where the top of the stack is on x86
- Stack frame layout on x86_64

Global variables, constant, and floating-point

Global variables are given an offset, the IDA should reference to it as `offset ADDR` . For constant variables of int, the compiler use the number directly in assembly code, this maybe due to the fact that it won't ever change and may reduce a few instructions. For float constant IDA refer to as `ds:dword_ADDR` . The value is a hex number with `single-precision/double-precision floating point` , to know if it is a float number, notice the instructions used with the number. They have `ss` for float or `sd` for double appended, `ss is Scalar Single-Precision` and `ds is Scalar Double-Precision` . A few instructions are `movess, divss, mulss, subss, addss, comiss` and there neighbors `sd` .

Floating-point numbers are not stored using general register `eax, ebx, ...` rather they use the four register for floating-point operation, `xmm0, xmm1, xmm2, xmm3` . And instructions that uses these register are floating-point specific instructions. However, when a casting occurs, `(float)int_var` , the variables must first be convert to either single-precision or double-precision floating-point. Such instructions are `CVT..2..` , where the missing parts are type to convert to:

- DQ : Packed Doubleword Integer
- PD : Packed Double-Precision Floating Point
- PS : Packed Single-Precision Floating Point
- PI : Packed Dword Integer
- TDQ : Trucated Packed Doubleword Integer
- TPD : Trucated Packed Double-Precision Floating Point
- TPS : Trucated Packed Single-Precision Floating Point
- TPI : Trucated Packed Dword Integer
- SD : Scalar Double-Precision Floating Point
- SS : Scalar Single-Precision Floating Point
- SI : Doubleword Integer

Right now we only are about Floating Point and Integer conversion, others are ignored.

The difference between Single-Precision and Double-Precision:

Single-Precision, approximately 7 decimal digits:

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
0 1      8 9                     31
```

and Double-Precision, Approximately 16 decimal digits:

```
S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0 1        11 12                                                63
```

Struct and Struct Array

Struct in C are defined as a collection of consecutive address. Every member need to match its alignment, if not a padding is used. The alignment for each member are:

- bool : 1
- char : 1
- short : 2
- int : 4
- float : 4
- double : 8

Alignment is important, because depend on how we arrange the members, the size of struct may change, and offset to. See geeksforgeeks/structure-member-alignment-padding-and-data-packing.

When reversing with IDA, we can use structure to define new structure and apply that structure to our variable, this way, we can have a better view of our decompiled code.

Exercise number 3, library:

Define the struct book as follows in Structures tab:

```
00000000 book            struc ; (sizeof=0x49, mappedto_37)
00000000                                        ; XREF: delete_book+A2/o
00000000                                        ; delete_book+13A/o
00000000 author_name     db 21 dup(?)           ; string(C)
00000015 book_title      db 51 dup(?)           ; string(C)
00000048 inUse           db ?                   ; XREF: delete_book+81/r
00000048                                        ; delete_book+11D/w
00000049 book            ends
```

Find `library` global variable where it is used, press `y`, and change the type and name to `book library[10]`. We will have a pretty print when decompile:

```
add_book() {
  // ...
  book = &library[counter];
  j_strcpy(library[counter].author_name, &author_name);
  j_strcpy(book->book_title, &book_title);
  book->inUse = 1;
  call_printf("The book is added with id: %d\n");
  // ...
}
```

Even in disassembly view:

```
; delete book
mov     eax, [ebp+b]
mov     [eax+book.inUse], 0
push    offset Source   ; Source
mov     eax, [ebp+b]
push    eax             ; Dest
call    j_strcpy
```

Templating

C++ has a way to template functions, and class. Templating makes programming easier by writing code that are more general. However, when compiled, the templating functions, and class will be split out to its own version.

Take a look back in our exercise number 2, array.

```
template<class T>
void gen_random(T* arr, size_t len);

template<class T>
void sort(T* arr, size_t len);

template<class T>
void print_array(T* arr, size_t len);
```

And we use these three functions for three types of `char, short, int`. The compiled result, are three different functions, as can be seen below, are the calls to `gen_random(T* arr, size_t len)`.

```
push    3               ; a2
lea     eax, [ebp+char_arr]
push    eax             ; a1
call    j_template_char_array_generation
add     esp, 8
push    3               ; a2
lea     eax, [ebp+word_arr]
push    eax             ; a1
call    j_template_word_array_generation
add     esp, 8
```

```
push    3               ; a2
lea     eax, [ebp+dword_arr]
push    eax             ; a1
call    j_template_dword_array_gener
```

The different part in three functions is only

```
; char
call    rand
add     esp, 8
mov     ecx, [ebp+arr]
add     ecx, [ebp+counter]
mov     [ecx], al           ; one byte

; short
call    rand
add     esp, 8
mov     ecx, [ebp+counter]
mov     edx, [ebp+arr]
mov     [edx+ecx*2], ax     ; two byte

; int
call    rand
add     esp, 8
mov     ecx, [ebp+counter]
mov     edx, [ebp+arr]
mov     [edx+ecx*4], eax    ; four byte
```

Class and virtual table pointer

Classes when compiled turns into struct and functions. For a normal class, without virtual functions, the layout will be just the same as struct, with functions call to itself is pass as `method(this*, ...)` . The `this` pointer is often save to `ecx` . So before calling out to functions, a `push ecx` will be made. Or if a function uses `ecx` without initialization, it is a method of a class. Member in class are just like in struct, they are ordered the same as when they were declared. But these are simple cases, a more common case will be classes inherit each other and virtual functions.

A recap to virtual functions:

```
class Animal {
public:
  virtual ~Animal() {}
  virtual void talk() {
    printf("Animal talk\n");
  }

  Animal() {}
};

class Cat : public Animal {
public:
  void talk() {
    printf("Meo Meo\n");
  }

  Cat() {}
};

class Dog : public Animal {
public:
  void talk() {
    printf("Gau gau\n");
  }

  Dog() {}
};
```

With main as

```
int main(int argc, char** argv) {
  Animal dog = Dog();
  Animal cat = Cat();
  Animal animal;

  dog.talk();
  cat.talk();
  animal.talk();
}
```

It will print

```
Animal talk
Animal talk
Animal talk
```

Why? Because Animal object has its own function table, and it will call that function. The compiler thinks it is an Animal, while it is not.

```
int main(int argc, char** argv) {
  Animal* dog = new Dog();
  Animal* cat = new Cat();
  Animal* animal = new Animal();

  dog->talk();
  cat->talk();
  animal->talk();
}
```

Now the result is correct, we have the three lines

```
Gau gau
Meo meo
Animal talk
```

Because now a pointer is made, when making a call, it will point to the specific class method through `virtual function table pointer` .

The struct for the three class lying under assembly could be:

```
; the class struct
+---+-------+
| 0 | vfptr |----v
+---+-------+    |
                 |
; vf table       V
+---+---------------+
| 0 | decons_addr   |--------->decons function
+---+---------------+
| 4 | talk_addr     |--------->talk function
+---+---------------+
```

Because of this, we often see a call to register, either `eax, ecx, esi` , or register offset. Because it will assign the virtual function table to the class.

Example of calling talk function, compiled on my `x86_64` machine using `g++ 8.3.0` .

```
; var_28h is Animal* dog
mov rax, qword [var_28h]        ; rax = dog address
mov rax, qword [rax]            ; move [dog+0] = vfptr to rax
add rax, 0x10                   ; rax = vfptr + 10
mov rax, qword [rax]            ; move [rax] = [vfptr+10] = talk_addr to rax
mov rdx, qword [var_28h]
mov rdi, rdx                    ; set up rdi = dog address
call rax                        ; call rax(rdi) === talk_function(this*)
```

You could verify the result on godbolt. It should be quite alike, the `vtable`, aka `virtual function table`, is on the output too, try inspect the values.

References:

- Dịch ngược chương trình hướng đối tượng
- Reversing C++ Virtual Functions 1 & 2
- Blackhat Reversing C++

Other compiler stuffs

When dividing a number to 10, the compiler do a magic operation to compute faster:

```
mov edx, 0xcccccccd
mov eax, [var]
mul edx
mov eax, edx
shr eax, 0x3

; eax = [var] / 10
```

Link for information cs.uaf.edu/cs301.

In some files I notice a `push 1` before calling `this*`. I don't know what this does, but just ignore it.

```
mov eax, [ecx]        ; this ptr
push 1                ; why?
call dword ptr [eax]  ; call eax(1) ??
```

Summary

Reversing a simple C, C++ program not using any STL data structure is quite easy. The only tricky part was mainly class and virtual function. In doing this exercise, I misunderstood my instructor and not using virtual function, on exercise number 4 bee. So I fixed and do another version of the bee using virtual function.

The compiled binaries and IDA packed database are located github/nganhkhoa/viettel_internship/week2. The IDA packed database for release version does not rename as much as the debug ones. It was because in the release build, many things were optimized, small functions doesn't get call, instead assembly version is expanded. Functions I discover so are `strcpy`, (`thuvien.exe`), and initialization of classes base of one same class, no call to self generate function (`bee_virtual.exe`). On release build, many variables are replaced completely by register, so register is heavily used to preserve stack space.