6.06 Initialize Player

Typically a DLL will have a general purpose. It might be a DLL used for setting up a linked list, logging information, graphics, etc. The DLL we are reversing does more than just one general task. This is because I wanted to introduce you to DLL reversing with some easier examples. I'm going to let you know that the `InitializePlayer`, `PrintPlayerStats`, and `MysteryFunc` functions are all related to each other. Again, this DLL is meant to be a learning tool. We will get into better examples later on.

With that said, let's get started.

It looks like there are two functions related to `Player`. `Player` is probably a structure (struct). To find out, we need to reverse the functions that use the `Player` struct and find out what it consists of. Let's start with `InitializePlayer` because it will most likely give us the most information. It should also let us know what the bare minimum is for a `Player` struct.

> Structs and classes are the same. The only difference is that one is private and one is public by default. Other than that, they both work exactly the same. It's fine to use class and struct interchangeably when talking about C++. C doesn't have classes, so if you're talking about C use struct instead of class.

Reversing

Function Signature:

Thanks to x64dbg's analysis, we can see the function declaration:

- `void __cdecl InitializePlayer(class Player * __ptr64)`

So the function returns void and takes a pointer to a `Player` class. If you didn't know already, when referencing the first element in a class/struct you do so with the base of the class. You can think of it like this:

- `Player` = First element.
- `Player` + X = Element at offset X. For example, if the first element is a 32-bit value, then the second element is at `Player` + 0x4 (32 bits).

Disassembly:

Let's take a look at the Assembly code and figure out what's going on.

Here is the InitializePlayer function disassembled:

```
00007FF930D01F2 MOV DWORD PTR DS:[RCX], 0x20
00007FF930D01F2 LEA RDX, QWORD PTR DS:[0x7FF930D39718]
00007FF930D01F2 MOV DWORD PTR DS:[RCX+0x4], 0x42C80000
00007FF930D01F3 MOV R8D, 0xB
00007FF930D01F3 ADD RCX, 0x8
00007FF930D01F3 JMP <dll.sub_7FF930D033C0>
```

- `MOV DWORD PTR DS:[RCX], 0x20` - This moves 0x20 (32 in decimal) into the first element of the new `Player` class. We now know that the first element in a `Player` is, most likely, an integer because of this.
- `LEA RDX, QWORD PTR DS:[0x7FF930D39718]` - This loads the address to the string "PLACEHOLDER" into RDX.
- `MOV DWORD PTR DS:[RCX + 0x4], 0x42C80000` - Moves 0x42C80000 into the second element in the `Player` object. We now know that the first element was a 4 byte integer. 0x42C80000 seems like a strange value. If you look at the value as a float it seems that it's 100. Since this function initializes a "Player", this could have some relation to a game. 100 is commonly used as a maximum health value. So for now we can assume 0x42C80000 is a float. To be sure, we would want to see if this value is ever handled with floating point registers (XMM#).

This next part is suspicious. It looks like some kind of function call, but it uses a jump.

- `MOV R8D, 0xB` - 0xB is 11 in decimal, which is also the length of the "PLACEHOLDER" string.
- `ADD RCX, 0x8` - 0x8 is added to RCX, As of now this doesn't make much sense, but as it will turn out this is to set RCX past the first two elements that have been initialized.

Let's follow the jump and see what happens.

Code Jumped To

Take a look at the code being jumped to from `InitializePlayer`:

```
00007FF930D033C PUSH RBX
00007FF930D033C PUSH RSI
00007FF930D033C PUSH RDI
00007FF930D033C PUSH R14
00007FF930D033C PUSH R15
00007FF930D033C SUB RSP, 0x20
00007FF930D033C MOV R14, QWORD PTR DS:[RCX+0x18]
00007FF930D033D MOV RSI, R8
00007FF930D033D MOV R15, RDX
00007FF930D033D MOV RBX, RCX
00007FF930D033D CMP R8, R14
00007FF930D033D JA dll.7FF930D03409
00007FF930D033D MOV RDI, RCX
00007FF930D033E CMP R14, 0x10
00007FF930D033E JB dll.7FF930D033EA
00007FF930D033E MOV RDI, QWORD PTR DS:[RCX]
00007FF930D033E MOV QWORD PTR DS:[RCX+0x10], RSI
00007FF930D033E MOV RCX, RDI
00007FF930D033F CALL <dll.sub_7FF930D0A4A0>
00007FF930D033F MOV RAX, RBX
00007FF930D033F MOV BYTE PTR DS:[RSI+RDI], 0x0
00007FF930D033F ADD RSP, 0x20
00007FF930D0340 POP R15
00007FF930D0340 POP R14
00007FF930D0340 POP RDI
00007FF930D0340 POP RSI
00007FF930D0340 POP RBX
00007FF930D0340 RET
```

The jump is definitely acting like a call. Actually, the previous section didn't have a prologue but this one does. That's a little odd, but compilers tend to do odd things.

Before we start reversing this, I want you to guess what's going on. The string is being passed, and what's probably the string length is being passed as well. I'd guess that, based on the name of the function, "InitializePlayer", this function is going to copy the string "PLACEHOLDER" to the new Player class. To do this, it will need to copy the string from one memory location to another. It might deal with allocating memory.

- Pushes - There are several pushes which are done to preserve registers.
- `MOV R14, QWORD PTR DS:[RCX + 0x18]` - Copies some data that is at offset 0x18 from the `Player` object. We don't know what Player+0x18 is right now.
- Some other moving is done that is pretty self-explanatory.
- `CMP R8, R14` - Compare R8 (0xB) to R14 (RCX + 0x18)
    - `JA dll.7FF930D03409` - JA (Jump if Above) is the same as JG (Jump if Greater) except it's used for unsigned. This jump is testing 0xB against whatever is at RCX + 0x18. You could debug a program that uses this DLL to find out what is at RCX + 0x18, but for now, let's just continue without knowing.

Let's peak at where this jump goes:

```
00007FF93790340 MOV RDI, 0x7FFFFFFFFFFFFFFF
00007FF93790341 CMP RSI, RDI
00007FF93790341 JA dll.7FF930D03505
```

0x7FFFFFFFFFFFFFFF is moved into RDI, and RSI is then compared to that. RSI is what we are guess is the length of the string. It then jumps if RSI is greater than RDI. Following that jump leads to a call, and following that call leads to the following:

```
00007FF93790592 SUB RSP, 0x28
00007FF93790592 LEA RCX, QWORD PTR DS:[0x7FF930D39790] ; "string too long"
00007FF93790592 CALL dll.7FF930D05EB8
```

That's nice to know! So that jump deals with error handling. This tells us that the code we were looking at is very likely to be some kind of copy or allocation. This also tells us that RCX+0x18 is likely some kind of maximum size for the string length.

Let's get back to where we were:

- `CMP R14, 0x10` - Compare R14 (RCX + 0x18) to 0x10 (16). Again, we don't know what's in RCX + 0x18. Since the previous jmp condition was a fail condition, we can assume that R8 is less than R14, so 0xB is less than RCX+0x18.
    - `JB dll.7FF930D033EA` - If RCX + 0x18 is below (less than) 16, then jump.
- `CALL <dll.sub_7FF930D0A4A0>` - The instructions before this call are interesting. If the previous jump (`JB dll.7FF930D033EA`) *is* taken, the first element in the player class, 32, is *not* copied into RDI. We can assume that this is making sure there's enough space for the string.

- `MOV QWORD PTR DS:[RCX+0x10], RSI` - RSI (0xB) is put into RCX + 0x10.
- `MOV RCX, RDI` - RDI (RCX from `InitializePlayer`, which is the `Player` structure) is moved into RCX. This restores RCX back to the original RCX instead of RCX + 8 caused by the `ADD RCX, 0x8` instruction in the first section of code.
- Another function is then called.

Since the rest of the function doesn't seem to be very interesting, let's follow that call and see what it does. Here is the function:

```
00007FF930D0A4A MOV R11, RCX
00007FF930D0A4A MOV R10, RDX
00007FF930D0A4A CMP R8, 0x10
00007FF930D0A4A JBE dll.7FF930D0A500
<snip>
```

I'm not showing the whole function, because it's not needed in this case.

- RCX (`Player`) is moved into R11.
- RDX ("PLACEHOLDER" string) is moved into R10.
- `CMP R8, 0x10` - Compares 0xB to 0x10.
    - `JBE dll.7FF930D0A500` - If R8 is below or equal to 0x10 then jump. **We know R8 is 0xB which is less than 0x10. So let's follow the jump.**

Following the jump:

```
00007FF930D0A50 MOV RAX, RCX
00007FF930D0A50 LEA R9, QWORD PTR DS:[0x7FF930D00000]
00007FF930D0A50 MOV ECX, DWORD PTR DS:[R9+R8*4+0x45000]
00007FF930D0A51 ADD RCX, R9
00007FF930D0A51 JMP RCX
```

- `MOV RAX, RCX` - Move the `Player` into RAX.
- `LEA R9, QWORD PTR DS:[0x7FF930D00000]` - If you follow 0x7FF930D00000 in the dump you'll see that this is the beginning of the binary. You can tell because of the "MZ" at the start.
- `MOV ECX, DWORD PTR DS:[R9 + R8 * 4 + 0x45000]` - If you have some experience, you'll know what's going on. R9 is the base of the binary. R8 is the size of our string. 0x45000 is some offset, probably an offset into some data storage area. In fact, if you do a bit of math and look at the memory mappings/section locations you will see that this is indeed an offset into the RDATA section.
- `ADD RCX, R9` - R9, the base of the binary, is then added to whatever was at R9 + R8 * 4 + 0x45000.
- `JMP RCX` - So this explains the previous few instructions. Those instructions must have been retrieving a function offset/address to call.

Now we need to find where the call goes, so let's do some math.

- R9 + R8*4 + 0x45000 = 0x7FF930D00000 + 0x2C + 0x45000 = 0x7FF930D4502C
- [0x7FF930D4502C] + 0x7FF930D00000 = A550 + 0x7FF930D00000 = 0x7FF930D0A550

A few quick notes. We need the value *at* 0x7FF930D4502C. In x64dbg you can right-click in a dump window, select "Go to", then choose "Expression". When you look at the value at 0x7FF930D4502C you see it's 0x50A5. However, because your architecture uses most likely uses little endian we need to flip it around to be A550. We are then left with the address 0x7FF930D0A550. Go to it by right-clicking in the disassembly window and select "Go to", then choose "Expression".

Here is disassembly of the new code we found:

```
00007FF930D0A55 MOV R8, QWORD PTR DS:[RDX]
00007FF930D0A55 MOVZX ECX, WORD PTR DS:[RDX+0x8]
00007FF930D0A55 MOVZX R9D, BYTE PTR DS:[RDX+0xA]
00007FF930D0A55 MOV QWORD PTR DS:[RAX], R8
00007FF930D0A55 MOV WORD PTR DS:[RAX+0x8], CX
00007FF930D0A56 MOV BYTE PTR DS:[RAX+0xA], R9B
00007FF930D0A56 RET
```

Well there you have it, the string being copied. I won't walk through the entire thing, it's just a series of copies.

We're not done yet! Follow the series of returns (which is very satisfying to do) back to the code jumped to by the original code:

```
00007FF930D033C PUSH RBX
00007FF930D033C PUSH RSI
00007FF930D033C PUSH RDI
00007FF930D033C PUSH R14
00007FF930D033C PUSH R15
00007FF930D033C SUB RSP, 0x20
00007FF930D033C MOV R14, QWORD PTR DS:[RCX+0x18]
00007FF930D033D MOV RSI, R8
00007FF930D033D MOV R15, RDX
00007FF930D033D MOV RBX, RCX
00007FF930D033D CMP R8, R14
00007FF930D033D JA dll.7FF930D03409
00007FF930D033D MOV RDI, RCX
00007FF930D033E CMP R14, 0x10
00007FF930D033E JB dll.7FF930D033EA
00007FF930D033E MOV RDI, QWORD PTR DS:[RCX]
00007FF930D033E MOV QWORD PTR DS:[RCX+0x10], RSI
00007FF930D033E MOV RCX, RDI
00007FF930D033F CALL <dll.sub_7FF930D0A4A0> ; WE JUST CAME FROM HERE
00007FF930D033F MOV RAX, RBX                ; WE ARE NOW HERE
00007FF930D033F MOV BYTE PTR DS:[RSI+RDI], 0x0
00007FF930D033F ADD RSP, 0x20
00007FF930D0340 POP R15
00007FF930D0340 POP R14
00007FF930D0340 POP RDI
00007FF930D0340 POP RSI
00007FF930D0340 POP RBX
00007FF930D0340 RET
```

Okay, so that's essentially the end of it. One last part I want to point out is the `MOV BYTE PTR DS:[RSI+RDI], 0x0` which adds a null byte to the end of the string. Which, if you don't know, is how strings are terminated.

Summary

- The first member is initialized to 0x20 (32 in decimal).
- The second member is a float, initialized to 100.
- Finally, the third member is a string initialized to "PLACEHOLDER". In order to initialize this member, a bunch of code is ran to copy the string from one location to another. We can assume that the function is something like this in source code:

```
class Player{
public:
  int member1;
  float health;
  string str; //Could be std::string, char*, can't be sure.
};

void InitializePlayer(Player* player){
  player->member1 = 32;
  player->health = 100.0f;
  player->str = "PLACEHOLDER";
}
```

But what about the initialization code for the string? Well, think about it, is a developer going to write their own initialization code? Probably not. What's more likely is that the string is a special data type in their programming language, and that code was pre-generated. For example, this would be the case for a `std::string` in C++.

Here is the actual source code (again, I have it because I wrote it):

```
class Player {
public:
    int score;
    float health;
    std::string name;
};

void InitializePlayer(Player* player) {
    player->score = 32;
    player->health = 100.0f;
    player->name = "PLACEHOLDER";
}
```

We nailed it!

Final Notes

That was a much more involved example than before. If you got lost, it's best to step through the code yourself. And remember, write comments and take notes! That was, honestly, a little confusing, so don't worry if you struggle. As always, I'm available for questions.

You may have noticed that the string copying code we saw had other code chunks around it that looked similar. Isn't that interesting...

<- Previous Lesson
Next Lesson ->

Chapter Home