Reversing.kr中的Adventure，怕是这里面最硬核的题了，因为最少人做出来。不过这主要是很多人不知道appx文件如何动态调试。（这个wp最终可能还是会被放出来，估计时间是首页被4470分占满时）

解题过程

程序安装

1. 需要将系统时间修改至证书时间，否则会有错误0x800B0101。
2. 安装前需要先安装证书
    1. 右键点击appx文件，选择“属性”->“数字签名”
    2. 点击“详细信息”，然后点击“查看证书”
    3. 选择“安装证书”->"本地计算机"->“将所有证书放入下列存储”->"受信任的根证书颁发机构"。
3. 安装前先安装dependency中的x86下文件，否则会说有冲突组件或不支持。
4. 运行时需要vccorlib110.dll和msvcp110.dll动态链接库。

调试配置

1. plm程序调试可以使用plmdebug.exe。这个是微软提供的调试工具，专门用于调试appx文件，可以通过winSDK获得。
2. 使用命令行将程序与调试器绑定。

    `plmdebug.exe /enableDebug <package> <调试器路径>`

    其中，package是plm程序的包名，一般与程序名称长很多，可以通过`plmdebug /query`查询，包名一般带有程序名称，所以比较好找。

    这里我使用的是一个叫x32dbg的调试器，因为OD调试时出现了不可解决的异常。
3. 将appx解压后可以得到一个exe文件，这个就是核心的二进制程序。虽然这个程序不可以运行，但可以用来分析逻辑。

分析

1. 运行程序，查看字符串，发现了两个明显的字符串“Flag1”和“Flag2”。追踪到对应位置，然后根据offset在IDA中查看反编译源码。

    这两个字符串出现在sub_403890中。经调试可以发现，这个函数就是打怪、计算flag、输出flag的关键函数。

    1. 打怪判断位置如下

        403be3-403c24为击中判断，首先判断有没有子弹，然后判断子弹位置与devil位置是否相同，如果同则jmp 3c8e。

```
while ( 1 )
{
  if ( *v23 )                                   判断有没有发射子弹
  {
    v24 = *(v9 - 60);
    if ( (float)(*(v22 - 1) + v72) > v24
      && (float)(v24 + v16) > *(v22 - 1)
      && (float)(*v22 + v73) > v20
      && (float)(v20 + v78) > *v22 )
    {                                           判断子弹和devil的位置
      break;
    }
  }
  ++v21;
  ++v23;                                        没发射子弹或没打中，devil前进
  v22 += 2;
  v75 = v21;
```

2. flag输出位置如下

一共有两个，打怪分数达到0xddb会输出flag1，打怪分数达到0x31159cd会输出flag2。

```
v26 = v4[46]-- != 0;                            devil剩余个数--
v4[47] = v26 + v4[47] - 1;
if ( *((_QWORD *)v4 + 23) == 0xAAAAAAAAAAA9CCFi64 )
{
  v4[36] ^= 0x98A96C7u;
  v4[37] ^= 0x5A0DC398u;
  v4[38] ^= 0xB773AABu;
  v4[39] ^= 0xBE8806A0;
  v4[40] ^= 0xD01612BE;
  v4[41] ^= 0x2873543Bu;
  v4[42] ^= 0x4461496Eu;
  v4[43] ^= 0x9C0BED2u;
  v27 = WindowsCreateStringReference(L"Flag1 is ", 9, &v61, &v89);
```

```
else if ( *((_QWORD *)v4 + 23) == 768614336353095901i64 )
{
  v4[36] ^= 0x646EB4ADu;
  v4[37] ^= 0xCA973FF2;
  v4[38] ^= 0xF5B9A83u;
  v4[39] ^= 0xEE15974E;
  v4[40] ^= 0x61A2771Bu;
  v4[41] ^= 0xDEB785ED;
  v4[42] ^= 0x52DBA003u;
  v4[43] ^= 0x9EBE0B4E;
  v81 = v4 + 52;
  v36 = sub_408980(L"Flag2 is ", 9, &v61);
```

3. 然后查找flag变量的引用就可以找到flag计算的位置。一共有两个：

```
v4[2 * (unsigned __int8)(signed int)v15 + 34] ^= *v85;
v4[2 * (unsigned __int8)(signed int)*v9 + 35] ^= *v25;
```

其中，v15 = *v9，v25 = v85，v4就是a1。很明显，这是一个面向对象开发的程序，小结中会说到。

```
v4[2 * (unsigned __int8)(signed int)*v9 + 34] = (*(int (__cdecl **)(_DWORD *, _DWORD, i
                                                  v4,
                                                  v4[2 * (unsigned __int8)v87 + 34],
                                                  v87);
v57 = (*(int (__cdecl **)(_DWORD *, _DWORD, _DWORD))(*v4 + 32))(
        v4,
        v4[2 * (unsigned __int8)(signed int)*v9 + 35],
        (unsigned __int8)(signed int)v9[1]);
v58 = (unsigned __int8)(signed int)*v9;
v91 = -1;
v4[2 * v58 + 35] = v57;
```

这个计算“隐藏”的很深，稍不留神就看不到了。其中，查看虚表可知v4 + 32是一个循环移位函数，相当于rol4(a2, a3)；v87 = *v9。所以这段代码翻译如下：

```
int i = (unsigned __int8)(signed int)*v9;
v4[2 * i + 34] = rol4(v4[2 * i + 34], i);//rol4就是将被移位参数看做4个字节的无符号整数
v4[2 * i + 35] = rol4( v4[2 * i + 35], (unsigned __int8)(*(int *)(v9+1));
```

2. 将打怪击中判断直接爆破掉，然后程序就可以自己运行出flag。

   经实践，得到第一个flag大概需要1h；经计算，得到第二个flag大概需要595天。另外，**flag是显示在score的位置，需要先下好断点，否则会被后续分数覆盖掉。**

   现在有两个思路：缩短打怪时间 或 破解flag计算原理。显然缩短打怪时间更容易，所以从这里入手。

patch打怪时间

1. 显然，打怪时间是受限于怪的释放时间的，所以只要怪不间断的释放，就能不间断的打怪。而怪的释放显然有一种随机性，所以程序中很可能使用rand函数来进行了控制。
2. 搜索rand函数的引用，找到了sub_4043a0函数。

```
int __cdecl sub_4043A0(int a1, int a2, int a3)
{
  return a2 + rand() % (a3 - a2 + 1);
}
```

3. 在动态调试中下断点，查看其引用，然后在IDA中查看对应伪代码。

```
(*(int (__cdecl **)(int, signed int, signed int))(v5 + 28))(a1, 1, 50) == 5
```

   由此可见，只要让这个函数永远返回5就可以不间断释放devil了。

4. 直接篡改sub_4043a0，使其返回值为5——结果程序崩溃。在动态调试中使用条件日志记录sub_4043a0的参数和返回值发现，该函数不止这一处调用。
5. 为将影响降低，对sub_4043a0进行了如下篡改。

```
.text:00404391                 sub     ecx, [ebp+arg_4]
                               cmp     ecx,6
                               jb      4043af
                               mov     eax,5
                               jmp     4043b8
.text:004043A0                 push    ebp
.text:004043A1                 mov     ebp, esp
.text:004043A3                 call    ds:rand
.text:004043A9                 mov     ecx, [ebp+arg_8]
.text:004043AC                 jmp     00404391 ;只是一个示例,应该jmp到[程序基址+4391]
.text:004043AF                 cdq
.text:004043B0                 inc     ecx
.text:004043B1                 idiv    ecx
.text:004043B3                 add     edx, [ebp+arg_4]
.text:004043B6                 mov     eax, edx
.text:004043B8                 pop     ebp
.text:004043B9                 retn
.text:004043B9 sub_4043A0      endp
```

6. 修改后就可以不间断的打怪了，score也飞速增长，很快就可以达到0xddb。虽然计算flag2还需要大概18天，但是已经是可以忍受的范围了。
7. 但是，得到的flag1已经不是原来那个flag1了，而是乱码。也就是说，4043a0函数的篡改会影响flag的生成。所以还是老老实实逆逻辑逆算法吧（*或者等上一年半，等别人都把前五十的坑占了，然后再优哉游哉地提交flag。期间还可能因为意外情况而重新运行程序，不过没关系，反正也不着急吗，呵呵*）。

破解flag计算原理

既然篡改4043a0对flag有影响，则说明flag的生成有rand函数的参与，反过来查找4043a0的使用可以破解出flag的计算。

1. 之前的分析中可以知道4043a0对应a1+0x1C，所以要查看a1+0x1C的调用。

   a1+0x1C的引用全部在一个if结构中，如下，分别对a1+0x4C4、a1+0x44C、a1+0x35C、a1+0x360、a1+0x26C五个数组和a1+0x53C、a1+0x540两个数据进行了操作：

```
if ( (*(int (__cdecl **)(int, signed int, signed int))(v5 + 0x1C))(a1, 1, 50) == 5
)
  {
    //a1 + 0x53C 从 0 增长到29，然后再从0开始增长，循环往复。作为一下操作的下标
    v7 = *(_DWORD *)(a1 + 0x53C);
    *(_DWORD *)(a1 + 0x53C) = v7 == 29 ? 0 : v7 + 1;

    if ( !*(_DWORD *)(a1 + 4 * *(_DWORD *)(a1 + 0x53C) + 0x4C4) )
    {
      *(float *)(a1 + 8 * *(_DWORD *)(a1 + 0x53C) + 0x26C) = (float)(*(int (__cdecl
**)(int, _DWORD, signed int))(*(_DWORD *)a1 + 0x1C))(
                                                                a1,
                                                                0,
                                                                (signed int)
(float)(a2

     - (float)(a4

             * *(float *)(a1 + 612))));
      *(_DWORD *)(a1 + 8 * *(_DWORD *)(a1 + 0x53C) + 0x270) =
COERCE_UNSIGNED_INT(a4 * *(float *)(a1 + 616)) ^ xmmword_40DC50;
      *(_DWORD *)(a1 + 4 * *(_DWORD *)(a1 + 0x53C) + 0x44C) = (*(int (__cdecl **)
(int, signed int, signed int))(*(_DWORD *)a1 + 0x1C))(
                                                            a1,
                                                            1,
                                                            2);
      *(float *)(a1 + 8 * *(_DWORD *)(a1 + 0x53C) + 0x35C) = (float)(*(int (__cdecl
**)(int, signed int, signed int))(*(_DWORD *)a1 + 0x1C))(
                                                                a1,
                                                                1,
                                                                4);
      *(float *)(a1 + 8 * *(_DWORD *)(a1 + 0x53C) + 0x360) = (float)(*(int (__cdecl
**)(int, signed int, signed int))(*(_DWORD *)a1 + 0x1C))(
                                                                a1,
                                                                4,
                                                                10);
      *(_DWORD *)(a1 + 4 * *(_DWORD *)(a1 + 0x53C) + 0x4C4) = 1;
      if ( *(_DWORD *)(a1 + 0x540) == -1 )
        *(_DWORD *)(a1 + 0x540) = *(_DWORD *)(a1 + 0x53C);
    }
  }
```

对于以上的数据，目前不知道是什么，也不知道意义何在，只好换个方向：从flag计算入手。

2. 通过之前分析可知，flag计算其实就是对v4[36]到v4[43]这8个unsigned int的异或和移位操作。操作的下标由v9控制，异或的数据为v85，移位的个数也是*v9和*(v9+1)。所以要找到v9和v85的来源。

   v9和v85的来源就在上面那个if函数的下面：

```
v9 = (float *)(a1 + 0x35C);
result = (_DWORD *)(a1 + 0x44C);
v82 = 0;
v76 = (float *)(a1 + 0x35C);
v85 = (_DWORD *)(a1 + 0x44C);
```

并且每次打怪后都会将v9和v85向后移动。

a1+0x35C和a1+0x44C就是那个if结构里面操作的数组。

3. 现在大概猜到了flag怎么来的了：以a1+0x35C数组元素为下标，以a1+0x44C数组元素为异或数、以a1+0x35C和a1+0x360数组元素为循环移位数进行计算。翻译如下：

```
int j = 0;
while(j < 30){
    int i = (a1+35c)[j]
    int x = (a1+44c)[j]
    v4[2 * i + 34] ^= x;
    v4[2 * i + 35] ^= x;

    int z = (a1+35c)[j]
    v4[2 * i + 34] = rol4(v4[2 * i + 34], z);

    z = (a1+360)[j]
    v4[2 * i + 35] = rol4(v4[2 * i + 35], z);

    j++;
}
```

4. 然后进一步详细分析可知：

a1+0x53C——怪物ID，用于记录新释放的怪物在数组中对应的下标，每个怪物相关的数组元素个数都为30，所以a1+0x53C从0到29进行循环。

a1+0x26C和a1+0x270——记录怪物的位置。

a1+0x4C4——记录怪物是否被消灭。

a1+0x540——记录当前激活的怪物ID

因此，flag计算逻辑如下：

```
//本来a1+0x540这些都是指针，这里只作为一个符号，就不管了
int id = 0;
while(id < 30){
    if((a1+0x4C4)[id] != 0 && id == (a1+0x540)){
            int i = (a1+35c)[id]
            int x = (a1+44c)[id]
            v4[2 * i + 34] ^= x;
            v4[2 * i + 35] ^= x;

            int z = (a1+35c)[id]
            v4[2 * i + 34] = rol4(v4[2 * i + 34], z);

            z = (a1+360)[id]
```

```
            v4[2 * i + 35] = rol4(v4[2 * i + 35], z);

            (a1+0x4C4)[id] = 0;//这个怪死了

            if((a1+0x540) == 29)//激活下一个怪
                (a1+0x540) = 0;
    }
    id++;
}
```

5. 接下来，只需要找到这些数据的初始值及填充。
    1. 初始化在sub_4032E0函数中，通过srand函数的引用就可以知道。

       a1+0x53C被初始化为-1，其余数据均初始化为0（包括flag）。
    2. 数据填充就在那个if结构中，用带有rand函数的sub_4043a0进行填充。

       众所周知，rand函数是一个伪随机函数，其实现原理为a = a * xxx +yyy，每个平台的具体实现有所不同。通过动态追踪可知，本程序中rand函数原理如下：

       ```
       unsigned int g_RandSeed = 0x64u; //srand的参数
       unsigned int rand() {
           unsigned int tmp = g_RandSeed;
           tmp = tmp * 0x343fd;
           tmp = tmp + 0x269ec3;
           g_RandSeed = tmp;
           tmp = tmp >> 0x10;
           tmp = tmp & 0x7fff;
           return tmp;
       }
       ```

6. 使用c语言实现flag计算：

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>
#include <tchar.h>

unsigned int g_RandTmp = 0x64;

unsigned int myrand() {//确定随机数生成与原程序一致
    unsigned int tmp = g_RandTmp;
    tmp = tmp * 0x343fd;
    tmp = tmp + 0x269ec3;
    g_RandTmp = tmp;
    tmp = tmp >> 0x10;
    tmp = tmp & 0x7fff;
    return tmp;
}

unsigned int rol(unsigned int t, int i) {
    unsigned int tmp = 0;
    tmp = (t >> (32 - i)) | (t << i);
```

```
    return tmp;
}

int main(int argc, char* argv[]) {
    unsigned int rand = 0;
    unsigned int flag[11] = { 0 };
    unsigned int a_44c[31] = { 0 }, a_35c[31] = { 0 }, a_360[30] = {0}, a_4c4[30] =
{ 0 };
    int a_540 = -1;//可以打哪个devil，必须按顺序打
    int v9, v85;
    int v8 = 0;

    int iDevilNum = -1, iShotDevil = 0;

    while(1) {
        //printf("%x\n", i);
        //get 5
        if ((myrand() % 50 + 1) == 5) {
            //get 53c
            iDevilNum = iDevilNum == 0x1d ? 0 : (iDevilNum + 1);//第几个devil
            if (a_4c4[iDevilNum] == 0) {//还没有这个devil
                //set local
                myrand();

                //set 44c
                a_44c[iDevilNum] = myrand() % 2 + 1;

                a_35c[iDevilNum] = myrand() % 4 + 1;

                a_360[iDevilNum] = myrand() % (10 - 4 + 1) + 4;

                a_4c4[iDevilNum] = 1;
                if (a_540 == -1)
                    a_540 = iDevilNum;
            }
        }

        v9 = a_35c[0];
        v85 = a_44c[0];
        int z = 0;

        do {//打一波怪

            if (a_4c4[z] == 1) {//有第z个怪
                if (z == a_540) {//这个怪是激活状态

                    flag[2 * v9] ^= v85;
                    flag[2 * v9 + 1] ^= v85;

                    iShotDevil++;//打怪个数
                    if (iShotDevil % 0x200000 == 0)
                        printf("shot 0x%x\n", iShotDevil);
                    if (iShotDevil == 0xddb) {
```

```
                    unsigned int * pUI = flag + 2;

                    pUI[0] ^= 0x98A96C7u;
                    pUI[1] ^= 0x5A0DC398u;
                    pUI[2] ^= 0xB773AABu;
                    pUI[3] ^= 0xBE8806A0u;
                    pUI[4] ^= 0xD01612BEu;
                    pUI[5] ^= 0x2873543B;
                    pUI[6] ^= 0x4461496E;
                    pUI[7] ^= 0x9C0BED2;

                    printf("flag1 : %s\n", flag + 2);
                }
                else if (iShotDevil == 0x31159cd)
                {
                    unsigned int * pUI = flag + 2;

                    pUI[0] ^= 0x646EB4ADu;
                    pUI[1] ^= 0xCA973FF2u;
                    pUI[2] ^= 0xF5B9A83u;
                    pUI[3] ^= 0xEE15974Eu;
                    pUI[4] ^= 0x61A2771Bu;
                    pUI[5] ^= 0xDEB785EDu;
                    pUI[6] ^= 0x52DBA003u;
                    pUI[7] ^= 0x9EBE0B4Eu;

                    printf("flag2 : %s", flag + 2);
                    system("pause");
                    return 0;
                }

                if (a_540 == 29)//激活下一个devil
                    a_540 = 0;
                else
                    a_540 = a_540 + 1;

                if (a_4c4[a_540] == 0)//如果没有下一个devil就恢复a_540初始值
                    a_540 = -1;

                flag[2 * v9] = rol(flag[2 * v9], v9);
                flag[2 * v9 + 1] = rol(flag[2 * v9 + 1], a_360[z]);

                a_4c4[z] = 0;//这个怪死了
            }
            //a_4c4[z] = 0;//这个怪死了
        }

        z++;//下个怪
        v9 = a_35c[z];
        v85 = a_44c[z];
    } while (z < 30);
}
```

```
    return 0;
}
```

小结

类成员函数的逆向

这个程序是面向对象开发的程序，在引用成员函数和成员变量时都是使用对象基址+偏移地址。

本程序还使用了很多虚函数，而虚函数地址都存在虚表中，那么在使用IDA分析时如何找到虚表？

1. 先找到一个函数(sub_xxx)
2. 通过动态调试的方法找到它的一个引用地址(yyy)
3. 在IDA中查找yyy对应的伪代码(a1+zzz)，这个a1就是虚表的起始地址。
4. 在IDA查找sub_xxx的存储地址(ooo)，这个地址就是(a1+zzz)的实际值
5. 那么虚表基址即为ooo - zzz

其实，a1就是类对象的this指针指向的地址。之所以虚表基址与a1相同是因为虚表存储在对象所在内存空间的起始位置。

x32dbg条件日志

x32dbg的条件日志编写与OD不太一样，具体可以参考OD 与X32 的条件记录断点的使用和对比

rand函数

rand函数是一个伪随机序列，只要种子相同，就会产生同样的一串数字。其原理如下：

```
a = a * xxx + yyy
```

其中，xxx和yyy都是常量，种子就是a的初始值。有些rand函数还会对计算出的a进行异或、移位等操作。

参考文章

- 错误代码0x800b0101解决办法
- appx程序证书安装
- OD 与X32 的条件记录断点的使用和对比