

☆

39

[原创] 某修仙肉鸽游戏协议逆向分析折腾记录  wuhuaguo888   1大侠   

37

17小时前

▲举报

459

4·

## 某修仙肉鸽游戏协议逆向分析折腾记录

前阵子无意中接触到ZMKS这款游戏，玩了两天后觉得挺有意思的，就想着能不能深入研究一下它的网络协议实现。这一折腾就是一多星期，踩了不少坑，但收获也挺大的。整个分析过程涉及到移动游戏逆向的很多典型场景，记录下来跟大家分享一下经验。

### 从APK开始的探索之旅

拿到APK后第一件事就是引擎识别，这步真的很关键。做过几个游戏分析的都知道，不同引擎的逆向套路完全不一样，走错了方向就是浪费时间。

解压APK直接奔向lib目录，几个关键文件一下子就映入眼帘：

- `libil2cpp.so` - 看到这个就知道是Unity IL2CPP编译的，基础引擎确定了
- `libxlua.so` - 这个更有意思，说明游戏逻辑是用Lua实现的
- `libmsaoaidsec.so` - 一看就是反调试保护，待会儿肯定要跟它斗智斗勇

 <code>libil2cpp.so</code>	18,189,664	65,964,384	SO 文件
 <code>libtuanjie.so</code>	9,076,536	21,874,264	SO 文件
 <code>libxlua.so</code>	1,852,261	5,165,992	SO 文件
 <code>libtapsdkcore.so</code>	1,708,644	4,605,352	SO 文件
 <code>libthemis.so</code>	490,505	1,038,680	SO 文件
 <code>libmsaoaidsec.so</code>	306,159	622,592	SO 文件
 <code>libCtaApiLib.so</code>	177,505	502,272	SO 文件
 <code>libmsaoaidauth.so</code>	77,902	376,832	SO 文件
 <code>libshadowhook.so</code>	116,197	317,848	SO 文件
 <code>libturingad.so</code>	140,974	308,616	SO 文件
 <code>libterrain.so</code>	84,845	256,176	SO 文件
 <code>libEncryptorP.so</code>	25,928	75,744	SO 文件
 <code>lib_burst_generated.so</code>	19,075	60,880	SO 文件
 <code>libluster.so</code>	18,023	39,568	SO 文件
 <code>libmain.so</code>	2,543	6,712	SO 文件

看到这个组合我心里就有数了。Unity作为渲染引擎负责底层，真正的游戏业务逻辑全在Lua脚本里。这种架构现在挺流行的，好处是热更新方便，但从逆向分析的角度来说，反而提供了更多的入口点。

### Unity IL2CPP层面的挖掘

既然确认是Unity引擎，那就直接上IL2CPPDumper了。这工具专门用来处理Unity的IL2CPP编译后的产物，能从native代码中还原出C#的类型信息和方法签名。

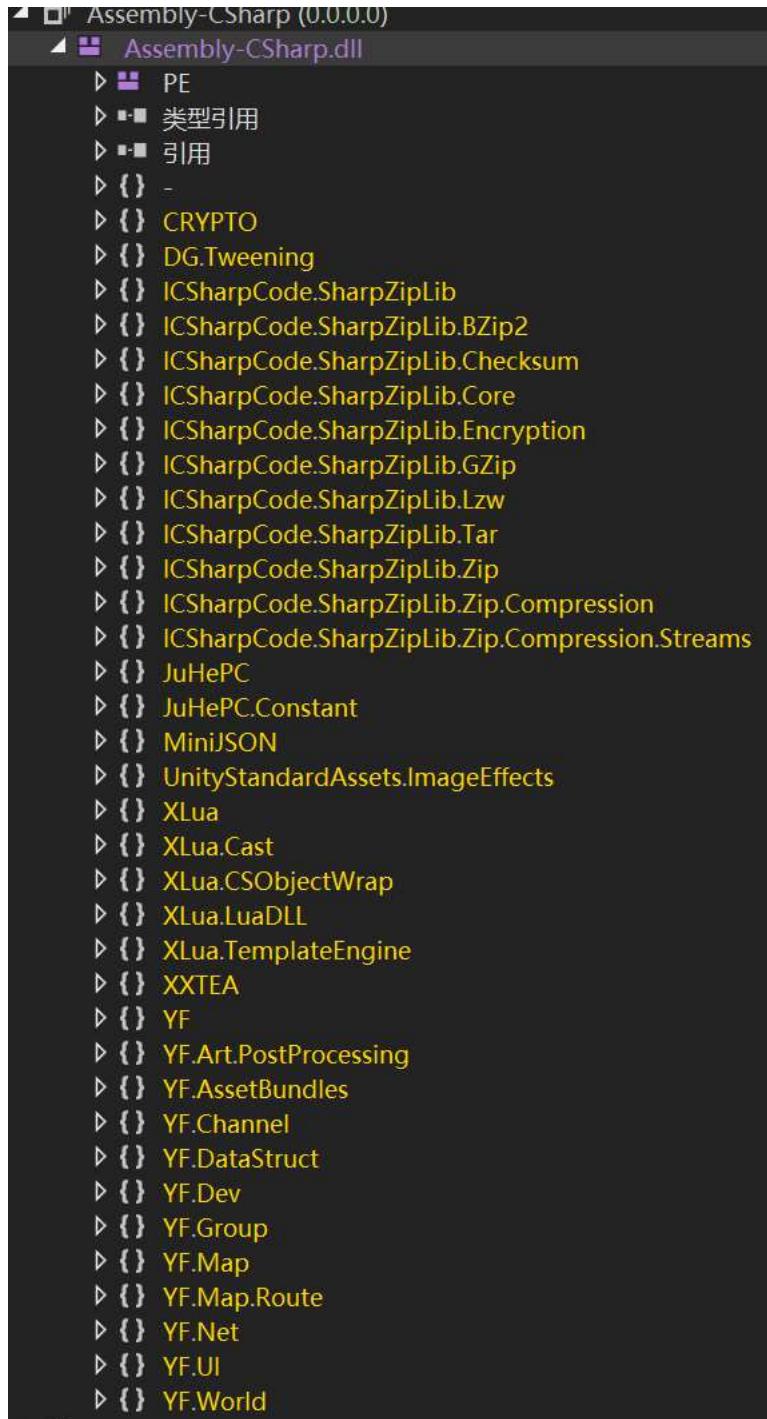
需要准备两个关键文件：

- `lib/arm64-v8a/libil2cpp.so` - 编译后的原生代码
- `assets/bin/Data/Managed/Metadata/global-metadata.dat` - 元数据信息

```
Initializing metadata...
Metadata Version: 31
Initializing il2cpp file...
Applying relocations...
WARNING: find JNI_OnLoad
ERROR: This file may be protected.
IL2Cpp Version: 31
Searching...
CodeRegistration : 3a89790
MetadataRegistration : 3bd4be8
Dumping...
Done!
Generate struct...
Done!
Generate dummy dll...
Done!
Press any key to exit...
```

☆ 39  
👍 37  
💬

运行IL2CPPDumper后，还好这个游戏没有在IL2CPP层面做保护，顺利dump出了所有的类型信息。用dnSpy打开生成的DLL文件一看，果然跟我预想的一样：



Assembly-CSharp.dll里几乎找不到什么游戏逻辑代码，大部分都是Unity基础类库和一些框架代码。这进一步印证了我的判断——游戏的核心业务逻辑确实在Lua层实现。

不过这里还是有几个有价值的发现：

- XLua相关的桥接代码，说明C#和Lua之间有完整的交互机制，这为后面的分析提供了线索
- YF.AssetBundles命名空间，这明显是游戏的资源管理系统，后面分析资源结构会用到
- 网络相关的基础类，虽然具体协议处理在Lua层，但底层socket还是C#实现的

### 跟反调试保护的第一次交锋

游戏集成了libmsaoaidsec.so这个反调试库，直接用Frida肯定会被检测到。这种保护挺常见的，主要检测这些：

1. 调试器特征 - ptrace、JDWP等调试接口的使用情况
2. Hook框架特征 - Frida、Xposed等工具的内存特征
3. 运行环境检测 - 模拟器、root环境的各种蛛丝马迹
4. 完整性校验 - APK签名和关键so文件的hash验证

碰到这种保护，一般有几种应对思路：

## 方案一：线程暂停

找到libmsaoaidsec.so创建的检测线程，直接暂停或kill掉。这种方法简单粗暴，但风险是可能影响游戏稳定性，有时候会莫名其妙崩溃。

## 方案二：函数patch

定位到具体的检测函数，用内存patch的方式将其NOP掉。这种方法效果最好，但工作量大，每个版本的libmsaoaidsec.so都需要重新分析。

39

## 方案三：去特征工具

使用修改过的Frida版本，比如Rusda，它移除了Frida的特征字符串，修改了默认端口等。

37

权衡了一下，我选择了Rusda这种方案。虽然可能随着保护升级会失效，但对付当前这个版本的libmsaoaidsec.so还是够用的，而且相对来说比较省事。

40

## 深入Lua脚本的世界

Unity+XLua架构中，所有基于Lua虚拟机的脚本加载都会经过`luaL_loadbufferx`这个函数。这是标准的Lua C API，对做过Lua逆向的人来说应该很熟悉：

```
1 int luaL_loadbufferx (
2     lua_State *L,           // Lua虚拟机状态
3     const char *buff,      // 脚本内容指针
4     size_t sz,             // 脚本内容大小
5     const char *name,      // 脚本文件名
6     const char *mode       // 加载模式 (text/binary/both)
7 );
```

通过Hook这个函数，理论上可以捕获到游戏动态加载的所有Lua脚本。写了个Frida脚本：

```
1 // Lua脚本动态捕获
2 function ensureDirectoryExists(filePath) {
3     const pathComponents = filePath.split('/').slice(0, -1);
4     let currentPath = '';
5
6     for (const component of pathComponents) {
7         currentPath += component + '/';
8         try {
9             const file = new File(currentPath, "r");
10            if (!file.exists()) {
11                file.close();
12                // 创建目录
13                const dir = new File(currentPath, "w");
14                dir.close();
15            } else {
16                file.close();
17            }
18        } catch(e) {
19            console.log("Directory creation error: " + e);
20        }
21    }
22 }
23
24 Interceptor.attach(Module.findExportByName('libxlua.so', "luaL_loadbufferx"), {
25     onEnter: function (args) {
26         const scriptName = args[3].readUtf8String();
27         const contentSize = args[2].toInt32();
28         const contentPtr = args[1];
29
30         // 只处理.lua文件
31         if (!scriptName || !scriptName.endsWith(".lua")) {
32             return;
33         }
34
35         const outputPath = "/sdcard/lua_analysis/" + scriptName;
36         ensureDirectoryExists(outputPath);
37
38         try {
39             const fileHandle = new File(outputPath, "wb");
40             const scriptContent = contentPtr.readByteArray(contentSize);
41
42             if (scriptContent) {
43                 fileHandle.write(scriptContent);
44                 fileHandle.flush();
45                 fileHandle.close();
46                 console.log(`[Lua Capture] ${scriptName} (${contentSize} bytes)`);
47             }
48         } catch(e) {
49             console.log(`[Error] Failed to save ${scriptName}: ${e}`);
50         }
51     });
52 });


```

但很快我就发现了这种方法的局限性：**只能获取游戏实际执行到的脚本**。游戏采用按需加载策略，很多功能模块可能根本不会被触发，这样就无法获取完整的脚本库。

要彻底解决这个问题，必须找到Lua脚本在APK中的实际存储位置，把完整的脚本库搞出来。

首页

社区

课程



招聘



发现

## AssetBundle资源系统的深度挖掘

现在的Unity游戏基本都用AssetBundle系统管理资源，这游戏也不例外。观察APK结构发现，`assets/AssetBundles`目录占据了绝大部分空间，显然游戏的核心资源都打包在这里。

名称	压缩后大小	原始大小	类型
..			
143702667c61f8a63a61510f22609a2e.assetbundle	18,033,856	18,033,856	ASSETBUNDLE 文件
01e03cfb9ce75fac667795e9370467a1.bytes	14,649,344	14,649,344	BYTES 文件
3211208ae4b5652329cc2d828e7f6bcb.assetbundle	11,688,522	11,688,522	ASSETBUNDLE 文件
9afb55910017eecc1a39650399b2a4cc.assetbundle	11,686,766	11,686,766	ASSETBUNDLE 文件
e28d7af8f3b2042bc14f2232307b4a9a.assetbundle	10,995,723	10,995,723	ASSETBUNDLE 文件
da976fa6699da21a9f399b2d52cbf067.assetbundle	10,949,646	10,949,646	ASSETBUNDLE 文件
e87bac3bb35e0f608af61e0da5c91499.assetbundle	10,942,732	10,942,732	ASSETBUNDLE 文件
7ddac5c10be31496302d319dd4e2e8807.assetbundle	10,880,433	10,880,433	ASSETBUNDLE 文件
e55a9701ec945b6117be6d9cef64ba70.assetbundle	10,860,995	10,860,995	ASSETBUNDLE 文件
33ae05f60bd042d557b605f338f9b066.assetbundle	10,844,813	10,844,813	ASSETBUNDLE 文件
5698d465e678eb9e40a543d95507b1ba.assetbundle	10,794,165	10,794,165	ASSETBUNDLE 文件
2afa721d45d8f16a90bf0a1b34605b39.assetbundle	10,787,099	10,787,099	ASSETBUNDLE 文件
baf9ede8e6c56d48d2ac424e80289dd3.assetbundle	10,749,522	10,749,522	ASSETBUNDLE 文件
49cf9b83d2c939b8b154e864379e858b.assetbundle	10,738,698	10,738,698	ASSETBUNDLE 文件
b60807563df68de0f730737776e65b9c.assetbundle	10,718,205	10,718,205	ASSETBUNDLE 文件
da9557f1f798e4f1738886e41febff2d.assetbundle	10,693,310	10,693,310	ASSETBUNDLE 文件
c00157df28bb8558bbe21ec214f28d41.assetbundle	10,617,925	10,617,925	ASSETBUNDLE 文件

对于做过Unity开发的人来说，AssetBundle并不陌生。它是Unity的模块化资源管理方案，有这些优势：

- **按需加载**: 只在需要时加载特定资源，节省内存
- **热更新支持**: 可以动态更新游戏内容而不用重新发布APK
- **平台优化**: 针对不同硬件平台优化资源格式
- **版本控制**: 方便管理不同版本的游戏资源

要找到Lua脚本的具体位置，关键是分析游戏的资源加载流程。从IL2CPP分析结果中，`AssetBundleManager`这个类很值得研究：

AssetBundleManager @02000685

- ▷ 基类和接口
- ▷ 派生类型
  - ⌚ AssetBundleManager() : void @06004F35
  - ⌚ AddAssetbundleAssetsCache(string) : bool @06004F26
  - \* AddAssetBundleCache(string, AssetBundle) : void @06004F1D
  - ⌚ AddAssetCache(string, Object[]) : void @06004F22
  - ⌚ CheckDependenciesAssetbundleByAssetbundleName(string, string, int) : void @06004F08
  - ⌚ Cleanup() : IEnumerator @06004F14
  - ⌚ ClearAssetsCache() : void @06004F12
  - \* CreateAssetBundleAsync(string, bool) : bool @06004F06
  - \* DecreaseReferenceCount(string) : int @06004F2F
  - \* Decryption(byte[], string) : byte[] @06004F02
  - ⌚ DownloadAssetBundleAsync(string, bool) : ResourceWebRequester @06004F0E
  - ⌚ DownloadAssetBundleAsync(GroupBox, string, bool) : ResourceWebRequester @06004F0F
  - ⌚ DownloadAssetFileAsync(GroupBox, string, bool, bool, bool) : ResourceWebRequester @06004F0G
  - ⌚ DownloadWebResourceAsync(string) : ResourceWebRequester @06004F0B
  - ⌚ GetAllDependenciesAssetbundleByAssetName(string) : List<string> @06004F07
  - ⌚ GetAssetBundleAsyncCreator(string) : ResourceWebRequester @06004F13
  - ⌚ GetAssetBundleCache(string) : AssetBundle @06004F1B
  - ⌚ GetAssetCache(string) : Object @06004F1F
  - ⌚ GetAssetCache(string, Type) : Object @06004F20
  - ⌚ GetAssetCache(string, string, Type) : Object @06004F21
  - ⌚ getDownloadUrl(GroupBox) : string @06004F17
  - \* GetReferenceCount(string) : int @06004F2C
  - \* IncreaseReferenceCount(string) : int @06004F2E
  - ⌚ InitializeGame() : IEnumerator @06004EFA
  - ⌚ InitializeStarter() : IEnumerator @06004EF9
  - \* InnerUnloadAssetBundle(string, bool, bool) : bool @06004F32
  - ⌚ IsAssetBundleLoaded(string) : bool @06004F1A
  - \* IsAssetBundleLock(string) : bool @06004F28
  - \* IsAssetBundleRecordLoaded(string) : bool @06004F2B
  - ⌚ IsAssetBundleResident(string) : bool @06004F19
  - ⌚ IsAssetLoaded(string) : bool @06004F1E
  - \* IsCodeAssetbundleName(string) : bool @06004F25
  - ⌚ IsFileInWebRequestingMap(string) : bool @06004F0D
  - ⌚ IsProsessingAssetBundleAsyncLoader(string) : bool @06004F0A
  - ⌚ LoadAssetAsync(string, Type) : BaseAssetAsyncLoader @06004EFC
  - ⌚ LoadAssetAsync(string, string, Type) : BaseAssetAsyncLoader @06004EFE
  - ⌚ LoadAssetAsyncVideo(string, Type) : BaseAssetAsyncLoader @06004EFD
  - ⌚ LoadAssetBundleAsync(string, bool, int) : BaseAssetBundleAsyncLoader @06004F09
  - ⌚ MapAssetPath(string, out string, out string) : bool @06004EFB
  - \* OnProsessingAssetAsyncLoader() : void @06004F05

这个类中的LoadAssetAsync方法是关键入口点。通过IDA Pro进行静态分析，可以还原出资源加载的完整逻辑：

```

YF_AssetBundles_BaseAssetAsyncLoader_o *YF_AssetBundles_AssetBundleManager__LoadAssetAsync(
    YF_AssetBundles_AssetBundleManager_o *this,
    System_String_o *assetPath,
    System_Type_o *assetType,
    const MethodInfo *method)
{
    const MethodInfo *method_5; // x4
    __int64 v8; // x1
    __int64 v9; // x1
    __int64 v10; // x1
    YF_AssetBundles_AssetAsyncLoader_o *item_1; // x0
    const MethodInfo *method_2; // x2
    System_Collections_Generic_List_object_o *processingAssetAsyncLoader; // x22
    struct System_Object_array *_items; // x8
    _QWORD *Method$System.Collections.Generic.List_AssetAsyncLoader_.Add(); // x9
    __int64 _size; // x10
    I12CppObject *item; // x21
    I12CppObject *message; // x19
    const MethodInfo_2AE6954 *method_1; // x0
    bool IsAssetLoaded; // w0
    const MethodInfo *method_3; // x3
    const MethodInfo *method_4; // x4
    System_String_o *assetName_1; // x22
    UnityEngine_Object_o *asset; // x0
    YF_AssetBundles_BaseAssetBundleAsyncLoader_o *loader; // x0
    System_String_o *assetName; // [xsp+8h] [xbp-38h] BYREF
    System_String_o *assetbundleName; // [xsp+18h] [xbp-28h] BYREF
}

if ( (byte_3EEEEA8 & 1) == 0 )
{
    sub_17D8B38(&YF_AssetBundles_AssetAsyncLoader_TypeInfo, assetPath);
    sub_17D8B38(&UnityEngine_Debug_TypeInfo, v8);
    sub_17D8B38(&Method_System_Collections_Generic_List_AssetAsyncLoader__Add__, v9);
    sub_17D8B38(&StringLiteral_8257, v10);
    byte_3EEEEA8 = 1;
}
assetbundleName = 0;
assetName = 0;
if ( YF_AssetBundles_AssetBundleManager__MapAssetPath(this, assetPath, &assetbundleName, &assetName, method_5) )
{
    // ...
}

```

总结下来这个函数的作用就是

1. 将 assetPath 映射为 (assetBundleName, assetName)
2. 取出/创建一个 AssetAsyncLoader，并放入“进行中加载列表”
3. 如果资源已在缓存，则直接完成该 loader；否则先异步加载对应的 AssetBundle

经过详细分析，我把资源加载流程总结为这几个步骤：

1. **路径映射阶段**：将逻辑资源路径（比如“luagame.assetpkg”）映射为物理文件路径
2. **缓存检查**：查看目标资源是否已经在内存缓存中
3. **异步加载**：如果缓存未命中，创建异步加载器从磁盘读取AssetBundle
4. **解密处理**：对加密的AssetBundle进行解密操作
5. **资源实例化**：将AssetBundle中的资源实例化为Unity对象

通过Hook `LoadAssetAsync`方法，我成功获得了资源名称与实际文件的映射关系：

```

1 Game.assetbundle -> 260051b7bf2af4070031708b056f55.assetbundle
2 gameassetsmap_bytes.assetbundle -> d1bd3d43c8bc6c1cd5a5dd34f9046e.assetbundle
3 gamedependencies_bytes.assetbundle -> 7a0c7f31bfe78603126a70cb97df4ae.assetbundle
4 luagame.assetpkg -> cced8de1b361f40750fbffcd0e046241.assetpkg # 这就是这个！
5 pb.assetbundle -> 09e9b1870b0bec20b4e772eaecd8831.assetbundle

```

看到`luagame.assetpkg`那一行，我心里一阵兴奋！这个文件应该就包含了游戏的完整Lua脚本库。对应的文件是`cced8de1b361f40750fbffcd0e046241.assetpkg`。

### 与加密机制的斗智斗勇

直接用AssetStudio打开目标文件，结果失败了。用十六进制编辑器检查，发现内容已经被加密：

cced8de1b361f40750fbffcd0e046241.assetpkg x																0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
39	1B	08	13	18	2B	36	2E	61	73	73	6D	41	5E	13	49	9	.	.	.	+6	.	a	s	m	A	^	I				
14	75	53	57	53	5F	4B	1D	4F	40	4B	11	47	70	6B	67	.	u	S	W	S	_K	.O	@	K	.	G	p	k	g		
6C	75	61	24	1C	A5	65	2E	63	09	73	65	71	FD	6B	67	l	u	a	\$	.	e	.	c	.	seq	.	k	g			
6E	36	61	67	61	6D	65	2E	61	73	73	65	74	70	6B	67	n	6	a	g	a	m	e	.	a	s	s	t	p	g		
72	75	60	67	D0	EB	65	2C	61	73	73	64	79	28	6B	64	r	u	`	g	.	.	,	a	s	s	d	y	(	k	o	
66	75	55	67	1E	22	6F	2E	45	0D	C4	6F	74	55	EE	6F	f	u	U	g	.	"	o	.	E	.	.	t	U	.		
72	75	75	2D	6B	6D	41	A7	49	79	73	41	09	C2	61	67	r	u	u	-	k	M	A	.	I	y	s	A	.	g		
48	03	9F	6D	61	49	1E	63	6B	73	57	ED	73	7A	6B	43	H	.	m	a	I	.	c	k	s	W	.	z	k	O		

文件头部没有标准的Unity AssetBundle签名，数据分布看起来完全是随机的，说明采用了某种加密算法。这下麻烦了，得想办法把密给破了。

继续分析AssetBundleManager类，终于找到了关键的解密方法。通过IDA的交叉引用功能，追踪到了Decryption函数：

```
// YF.AssetBundles.AssetBundleManager  
// Token: 0x06004F02 RID: 20226 RVA: 0x00002050 File Offset: 0x00000250  
[Token(Token = "0x6004F02")]  
[Address(RVA = "0x2256404", Offset = "0x2255404", VA = "0x2256404")]  
private byte[] Decryption(byte[] data, string psw)  
{  
    return null;  
}
```

深入分析解密逻辑后，发现用的是最简单的XOR异或加密：

```
do  
{  
    if (_stringLength >= psd->fields._stringLength )  
        index = 0;  
    else  
        index = _stringLength;  
    if ( len_2 >= LODWORD(data->max_length) )  
        sub_17D8D6C();  
    v11 = p_m_Items[len_2];  
    _stringLength = index + 1;  
    p_m_Items[len_2++] = v11 ^ System_String_get_Chars(psd, index, 0);  
}  
while ( len_1 != len_2 );
```

XOR加密的特点是简单高效，加密和解密使用相同的算法：`encrypted_data[i] = original_data[i] ^ key[i % key_length]`

现在最关键的问题是：密钥到底是什么？

通过向上追踪函数调用链，在更高层的调用中我找到了答案：

```
1 // 关键代码片段的逆向还原  
2 if (webRequester->isEncryptionEnabled) {  
3     byte[] encryptedData = webRequester.GetBytes();  
4  
5     // 解密调用：数据 + AssetBundle名称作为密钥  
6     byte[] decryptedData = AssetBundleManager.Decryption(  
7         encryptedData, // 加密数据  
8         webRequester.assetBundleName // 密钥竟然就是文件名！  
9     );  
10  
11     AssetBundle bundle = AssetBundle.LoadFromMemory(decryptedData);  
12 }
```

看到这里我都笑了，密钥竟然就是AssetBundle的文件名。这种设计虽然简单，但对于防止普通用户随意修改资源确实有一定效果。

有了密钥，写解密脚本就很简单了：

```

1  def decrypt_assetbundle_xor(encrypted_data: bytes, key_string: str) -> bytes:
2      """
3          AssetBundle XOR解密实现
4
5      Args:
6          encrypted_data: 加密的字节数据
7          key_string: 密钥字符串（通常为AssetBundle文件名）
8
9      Returns:
10         解密后的字节数据
11     """
12     key_bytes = key_string.encode('utf-8')
13     key_length = len(key_bytes)
14
15     decrypted_data = bytearray()
16
17     for i, encrypted_byte in enumerate(encrypted_data):
18         key_byte = key_bytes[i % key_length]
19         decrypted_byte = encrypted_byte ^ key_byte
20         decrypted_data.append(decrypted_byte)
21
22     return bytes(decrypted_data)
23
24     # 实际解密过程
25     with open('cced8de1b361f40750fbbfc0e046241.assetpkg', 'rb') as f:
26         encrypted_content = f.read()
27
28     decrypted_content = decrypt_assetbundle_xor(encrypted_content, 'luagame.assetpkg')
29
30     # 验证解密结果
31     if decrypted_content.startswith(b'UnityFS'):
32         print("解密成功！文件格式: UnityFS")
33         with open('luagame_decrypted.assetbundle', 'wb') as f:
34             f.write(decrypted_content)
35     else:
36         print("解密失败，请检查密钥")

```

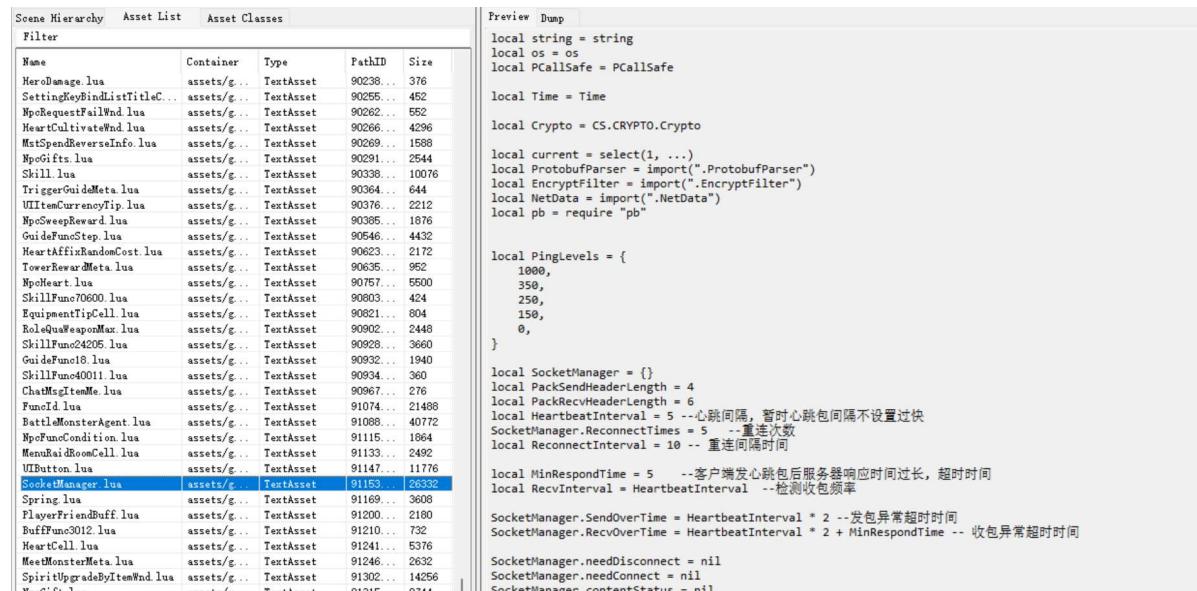
解密成功后，文件显示为标准的UnityFS格式：

```

55 6E 69 74|79 46 53 00|00 00 00 08|35 2E 78 2E|UnityFS.....5.x
78 00 32 30|32 32 2E 33|2E 33 38 74|33 00 00 00|x.2022.3.38t3...

```

用AssetStudio一解析，整个Lua脚本库都出现在眼前：



看到这个结果，我心里那个激动啊！经过这么多轮的分析和破解，终于拿到了游戏的完整源码。

## 网络协议的庐山真面目

有了完整的Lua源码，协议分析就变得清晰多了。从代码目录结构可以看出，网络相关的代码主要集中在几个关键文件里：

先看看客户端协议ID定义 (`net/ccmd.lua`)：

```

1 local pb = require("pb")
2
3 local function enum(id)
4     return pb.enum("com.yofijoy.core.proto.CSProtoId", id)
5 end
6
7 local CCmd = {}
8
9 CCmd.HEARTBEAT = enum("CS_HEART_REQ") --心跳
10
11 -- 登录相关
12 CCmd.USER_LOGIN = enum("CS_USER_LOGIN_REQ") -- 用户登录
13 CCmd.CREATE_ROLE = enum("CS_CREATE_ROLE_REQ") -- 创建角色
14 CCmd.ROLE_LOGIN = enum("CS_ROLE_LOGIN_REQ") -- 角色登录
15 CCmd.ROLE_OPERATION = enum("CS_ROLE_OPERATION_REQ") -- 操作角色
16 CCmd.USER_LOGIN_ACTIVATE_CODE_RPT = enum("CS_USER_LOGIN_ACTIVATE_CODE_RPT")-- 使用激活码登录

```

再看看服务端协议ID定义 (net/scmd.lua) :

```

1 local pb = require("pb")
2
3 local function enum(id)
4     return pb.enum("com.yofijoy.core.proto.CSProtoId", id)
5 end
6
7 local SCmd = {}
8
9 SCmd.HEARTBEAT = enum("CS_HEART_RESP") --心跳返回
10
11 SCmd.USER_LOGIN = enum("CS_USER_LOGIN_RESP") -- 用户登录返回
12 SCmd.CREATE_ROLE = enum("CS_CREATE_ROLE_RESP") -- 创建角色返回
13 SCmd.ROLE_LOGIN = enum("CS_ROLE_LOGIN_RESP") -- 角色登录

```

然后是Protobuf序列化处理 (net/ProtobufParser.lua) :

```

1 local current = select(1, ...)
2 local EncryptFilter = import(".EncryptFilter")
3 local pb = require "pb"
4 local ProtobufParser = {}
5
6 local PbBytes =
7     {
8         "Pb/base.bytes",
9         "Pb/CSProto.bytes",
10    }
11
12 function ProtobufParser:coInit()
13     -- 协程请求协议二进制文件
14     for i, byteFile in ipairs(PbBytes) do
15         local asset = ResourcesManager:coLoadAsync(byteFile, typeof(UE.TextAsset))
16
17         if not asset then
18             assert(false, "protobuf 资源异步加载失败")
19             break
20         end
21
22         local ret = pb.load(asset.bytes)
23         if ret == false then
24             assert(false, "protobuf 二进制数据加载失败")
25             break
26         end
27
28         -- print("加载", byteFile, "成功")
29     end
30
31 end

```

还有网络通信管理器 (net/LunJianSocketManager.lua) , 这个文件里有个有意思的发现:

```

1 local HeartbeatInterval = 5 --心跳间隔, 暂时心跳包间隔不设置过快
2 local ReconnectTimes = 5 --重连次数
3 local ReconnectInterval = 10 -- 重连间隔时间
4
5 local MinRespondTime = 5 --客户端发心跳包后服务器响应时间过长, 超时时间
6 local RcvInterval = HeartbeatInterval --检测收包频率
7
8 LunJianSocketManager.SendOverTime = HeartbeatInterval * 2 --发包异常超时时间
9 LunJianSocketManager.RcvOverTime = HeartbeatInterval * 2 + MinRespondTime -- 收包异常超时时间
10
11 local defaultEncryptKey = "spqh4hpstriat0q9h" -- 这个很关键!
12 LunJianSocketManager.encryptKey = defaultEncryptKey
13
14 local SocketError =
15     {
16         NORMAL = 0, --正常关闭, 在连接前也会关一下
17         ERROR_1 = -1, --C# send线程处理出现异常, 访问已释放的对象
18         ERROR_2 = -2, --C# send线程处理出现 发送数据包, 出现的非访问释放对象异常
19         ERROR_3 = -3, --C# recv线程 连接服务器或者接收到服务器数据读取不到数据, 会产生访问已释放的对象异常, 证明服务器已经关闭链接了
20         ERROR_4 = -4, --C# recv线程 连接或者关闭socket, 或接收数据包读取, 出现的非访问释放对象异常
21         ERROR_5 = -5, --主动断开连接, (重连重连, 非主动断开.忽略该信号)
22         ERROR_6 = -6, --主动连接超时
23     }

```

看到那个`defaultEncryptKey = "spqh4hpstriat0q9h"`, 我差点笑出声。这就是网络协议的AES加密密钥! 直接硬编码在脚本里, 简单粗



## 协议格式的真相大白

仔细分析了 `LunJianSocketManager.lua` 中的发送和接收逻辑，终于搞清楚了游戏网络协议的完整格式。

发送逻辑是这样的：

```
1 function LunJianSocketManager:send(msgCmd, msgData)
2     local protoData = nil
3     local msgLen = PackSendHeaderLength -- 发送包头长度常量
4
5     -- Step 1: Protobuf序列化
6     if msgData then
7         protoData = ProtobufParser:encode(msgCmd, msgData)
8         if not protoData then
9             LogError("Protobuf编码失败: " .. tostring(msgCmd))
10            return false
11        end
12    end
13
14    -- Step 2: 条件加密处理
15    if protoData then
16        -- 检查协议是否需要加密
17        if EncryptFilter:needEncrypt(msgCmd) then
18            protoData = Crypto.AesEncryptECB(protoData, self.encryptKey)
19            if not protoData then
20                LogError("AES加密失败: " .. tostring(msgCmd))
21                return false
22            end
23        end
24        msgLen = msgLen + #protoData
25    end
26
27    -- Step 3: 构建网络数据包
28    self.netData:reset()
29    self.netData:writeUShort(msgLen)      -- 包总长度 (2字节)
30    self.netData:writeUShort(msgCmd)      -- 协议ID (2字节)
31
32    if protoData then
33        self.netData:writeBuffer(protoData) -- 消息体数据
34    end
35
36    -- Step 4: 网络发送
37    return self:sendRawPacket(self.netData:getBuffer())
38 end
```

接收逻辑稍有不同：

```
1 function LunJianSocketManager:onProcessMsg(rawBytes)
2     self.netData:setBuffer(rawBytes)
3
4     -- 解析包头信息
5     local totalLength = self.netData:readInt()          -- 包总长度 (4字节)
6     local protocolId = self.netData:readUShort()        -- 协议ID (2字节)
7
8     local dataLength = totalLength - PackRecvHeaderLength
9
10    -- 读取消息体
11    local protobufData = self.netData:readProtocolBuffer()
12
13    if string.len(protobufData) ~= dataLength then
14        LogError("协议数据长度不匹配: expected=" .. dataLength .. ", actual=" .. string.len(protobufData))
15        return false
16    end
17
18    -- Protobuf反序列化
19    local messageData = ProtobufParser:decode(protocolId, protobufData)
20    if not messageData then
21        LogError("Protobuf解码失败: " .. tostring(protocolId))
22        return false
23    end
24
25    -- 消息分发处理
26    self:dispatchMessage(protocolId, messageData)
27    return true
28 end
```

通过源码分析，游戏协议格式的关键特征总结如下：

**客户端发送格式：**

```
1 [包长度:2字节] + [协议ID:2字节] + [消息数据:变长,可能AES加密]
```

**服务端响应格式：**

```
1 [包长度:4字节] + [协议ID:2字节] + [消息数据:变长,明文Protobuf]
```

这里有几个有意思的设计差异：

1. 发送和接收的包长度字段大小不同 (2字节 vs 4字节)
2. 发送的消息体可能进行AES-EBCB加密，接收的消息体是明文

3. 加密策略由`EncryptFilter:needEncrypt()`控制，不是所有协议都加密

### Protobuf协议定义的逆向重建

游戏使用lua-protobuf库处理消息序列化，协议定义的加载方式是这样的：

```
1 -- ProtobufParser.lua 核心逻辑
2 local pb = require("pb")
3
4 -- 预编译的protobuf定义文件
5 local ProtobufBinaryFiles = {
6     "Pb/base.bytes",           -- 基础消息类型定义
7     "Pb/CSProto.bytes",       -- 客户端-服务端协议定义
8 }
9
10 function ProtobufParser:initialize()
11     -- 异步加载二进制protobuf定义
12     for _, binaryFile in ipairs(ProtobufBinaryFiles) do
13         local asset = ResourcesManager:loadAssetSync(binaryFile, typeof(UE.TextAsset))
14
15         if asset and asset.bytes then
16             local success = pb.load(asset.bytes)
17             if not success then
18                 LogError("Failed to load protobuf definition: " .. binaryFile)
19                 return false
20             end
21         else
22             LogError("Protobuf definition file not found: " .. binaryFile)
23             return false
24         end
25     end
26
27     LogInfo("Protobuf definitions loaded successfully")
28     return true
29 end
```

这里有个问题：`lua-protobuf`使用的是预编译的.pb二进制文件，而不是可读的.proto源文件。要想还原出完整的协议定义，需要想办法从二进制格式逆向出可读的文本格式。

好在`lua-protobuf`提供了强大的反射机制，可以在运行时查询内存中的协议定义信息。

<code>pb.types()</code>	<code>iterator</code>	遍历内存数据库里所有的消息类型，返回具体信息
<code>pb.type(type)</code>	详情见下	返回内存数据库特定消息类型的具体信息
<code>pb.fields(type)</code>	<code>iterator</code>	遍历特定消息里所有的域，返回具体信息
<code>pb.field(type, string)</code>	详情见下	返回特定消息里特定域的具体信息
<code>pb.field(type, number)</code>	详情见下	返回特定消息里特定域的具体信息

利用这些反射接口，可以写个协议信息提取器：



39



37



```
1  -- Protobuf协议信息提取器
2  function extract_protobuf_schema()
3      local schema_database = {}
4      local enum_types = {}
5      local message_types = {}

6      -- 遍历所有类型定义
7      for full_typename, base_typename, type_kind in pb.types() do
8          local type_entry = {
9              full_name = full_typename,
10             base_name = base_typename,
11             type_kind = type_kind, -- "enum" or "message"
12             fields = {},
13             package = extract_package_name(full_typename)
14         }

15        -- 提取字段定义信息
16        for field_name, field_number, field_type, default_val, field_flags, oneof_name, oneof_idx in pb.fields(full_typename) do
17            local field_entry = {
18                name = field_name,
19                number = field_number,
20                type = field_type,
21                default_value = default_val,
22                flags = field_flags, -- "optional", "required", "repeated"
23                oneof_name = oneof_name,
24                oneof_index = oneof_idx
25            }

26            type_entry.fields[field_number] = field_entry
27        end

28        schema_database[full_typename] = type_entry
29    end

30    schema_database[full_typename] = type_entry

31    -- 按类型分类
32    if type_kind == "enum" then
33        enum_types[full_typename] = type_entry
34    elseif type_kind == "message" then
35        message_types[full_typename] = type_entry
36    end
37
38    end
39

40    return {
41        all_types = schema_database,
```

```
42    }
43}

44
45
46
47
48 function extract_package_name(full_typename)
49     local parts = {}
50     for part in string.gmatch(full_typename, "[^%.]+") do
51         table.insert(parts, part)
52     end

53

54     if #parts > 1 then
55         -- 移除最后一个部分（类型名），剩下的是包名
56         table.remove(parts, #parts)
57         return table.concat(parts, ".")
58     else
59         return ""
60     end
61 end
62
```

通过这个提取器，拿到了详细的协议信息JSON数据。然后又写了Python脚本将其重建为标准的.proto文件：

```

1  def rebuild.protobuf_definition(schema_data: dict) -> str:
2      """
3          将提取的协议信息重建为.proto文件格式
4      """
5      proto_lines = [
6          'syntax = "proto3";',
7          '',
8      ]
9
10     # 按包名组织类型定义
11     packages = {}
12     for type_name, type_info in schema_data['all_types'].items():
13         package_name = type_info.get('package', '')
14         if package_name not in packages:
15             packages[package_name] = {'enums': [], 'messages': []}
16
17         if type_info['type_kind'] == 'enum':
18             packages[package_name]['enums'].append(type_info)
19         elif type_info['type_kind'] == 'message':
20             packages[package_name]['messages'].append(type_info)
21
22     # 生成各个包的定义
23     for package_name, types in packages.items():
24         if package_name:
25             proto_lines.append(f'package {package_name};')
26             proto_lines.append('')
27
28     # 生成枚举定义
29     for enum_info in types['enums']:
30         proto_lines.extend(build_enum_definition(enum_info))
31         proto_lines.append('')
32
33     # 生成消息定义
34     for message_info in types['messages']:
35         proto_lines.extend(build_message_definition(message_info, package_name))
36         proto_lines.append('')
37
38     return '\n'.join(proto_lines)
39
40 def build_enum_definition(enum_info: dict) -> list:
41     lines = [f"enum {enum_info['base_name']} {{"]
42
43     # 按字段编号排序
44     sorted_fields = sorted(enum_info['fields'].items(), key=lambda x: int(x[0]))
45
46     for field_num, field_info in sorted_fields:
47         lines.append(f"    {field_info['name']} = {field_info['number']};")
48
49     lines.append("}")
50     return lines
51
52 def build_message_definition(message_info: dict, package_name: str) -> list:
53     lines = [f"message {message_info['base_name']} {{"]
54
55     # 按字段编号排序
56     sorted_fields = sorted(message_info['fields'].items(), key=lambda x: int(x[0]))
57
58     for field_num, field_info in sorted_fields:
59         field_type = normalize_field_type(field_info['type'], package_name)
60         field_prefix = ""
61
62         # 处理repeated字段
63         if field_info['flags'] == 'repeated':
64             field_prefix = "repeated "
65
66         lines.append(f"    {field_prefix}{field_type} {field_info['name']} = {field_info['number']};")
67
68     lines.append("}")
69     return lines
70
71 def normalize_field_type(original_type: str, current_package: str) -> str:
72     """规范化字段类型名称，处理包引用"""
73     if original_type.startswith('.'):
74         # 绝对路径类型引用
75         if original_type.startswith(f'.{current_package}.'):
76             # 同包引用，移除包前缀
77             return original_type[len(f'.{current_package}.'):]
78         else:
79             # 跨包引用，保留相对路径
80             return original_type[1:] # 移除开头的点
81     else:
82         # 基础类型或相对引用
83         return original_type

```

经过这一番折腾，终于成功重建出了完整的.proto文件，包括协议ID枚举：

```
1 |     syntax = "proto3";
2 | 
3 | package com.yofijoy.core.proto;
4 | 
5 | // 客户端-服务端协议ID枚举
6 | enum CSProtoId {
7 |     CS_FIRSTID = 0;
8 | 
9 |     // 基础协议
10 |     CS_HEART_REQ = 10001;           // 心跳请求
11 |     CS_HEART_RESP = 10002;         // 心跳响应
12 |     CS_USER_LOGIN_REQ = 10003;     // 登录请求
13 |     CS_USER_LOGIN_RESP = 10004;    // 登录响应
14 |     CS_CREATE_ROLE_REQ = 10005;    // 创建角色请求
15 |     CS_CREATE_ROLE_RESP = 10006;   // 创建角色响应
16 | 
17 |     // 游戏功能协议
18 |     CS_NPC_DOUBLE_CULTIVATE_REQ = 11125; // NPC双修请求
19 |     CS_NPC_DOUBLE_CULTIVATE_RESP = 11126; // NPC双修响应
20 |     // ... 更多协议定义
21 | }
22 | 
23 | // NPC双修请求消息结构
24 | message CS_NpcDoubleCultivateReq {
25 |     uint32 objid = 1;             // 对象ID
26 |     uint32 type = 2;              // 双修类型
27 |     bool tenTimes = 3;            // 是否进行十倍操作
28 | }
29 | 
30 | // NPC双修响应消息结构
31 | message CS_NpcDoubleCultivateResp {
32 |     uint32 result_code = 1;        // 操作结果码
33 |     string result_message = 2;     // 结果描述信息
34 |     RewardInfo rewards = 3;        // 获得的奖励信息
35 | }
```

## 实战验证：真刀真枪解数据包

现在万事俱备，该验证一下分析结果的正确性了。拿个实际抓到的数据包来测试：

```
1 | 1400752b05e3ce1f940f618eb295ea6c6c9c26cc
```

按照分析的协议格式，写了个完整的解析程序：

```

1 import struct
2 from Crypto.Cipher import AES
3 import CS_NpcDoubleCultivateReq_pb2 # 生成的protobuf类
4
5 def parse_game_packet(hex_data: str):
6     """解析游戏网络数据包"""
7     packet_bytes = bytes.fromhex(hex_data)
8
9     # Step 1: 解析包头
10    packet_length = struct.unpack('<H', packet_bytes[0:2])[0] # 小端序2字节
11    protocol_id = struct.unpack('<H', packet_bytes[2:4])[0] # 小端序2字节
12    message_data = packet_bytes[4:]
13
14    print(f"数据包长度: {packet_length}")
15    print(f"协议ID: {protocol_id} (0x{protocol_id:04x})")
16    print(f"消息体长度: {len(message_data)}")
17
18    # Step 2: 协议ID映射
19    protocol_mapping = {
20        11125: ("CS_NPC_DOUBLE_CULTIVATE_REQ", "CS_NpcDoubleCultivateReq"),
21        11126: ("CS_NPC_DOUBLE_CULTIVATE_RESP", "CS_NpcDoubleCultivateResp"),
22        # ... 更多映射
23    }
24
25    if protocol_id not in protocol_mapping:
26        print(f"未知协议ID: {protocol_id}")
27        return None
28
29    protocol_name, message_class = protocol_mapping[protocol_id]
30    print(f"协议名称: {protocol_name}")
31    print(f"消息类型: {message_class}")
32
33    # Step 3: 数据解密 (如果需要)
34    decrypted_data = message_data
35    if is_encrypted_protocol(protocol_id):
36        # 这里使用从游戏中提取的AES密钥
37        encryption_key = "spqh4hpstria0q9h".encode('utf-8')[16] # AES-128需要16字节密钥
38        decrypted_data = aes_decrypt_ecb(message_data, encryption_key)
39        print("数据已解密")
40
41    # Step 4: Protobuf反序列化
42    try:
43        if message_class == "CS_NpcDoubleCultivateReq":
44            message_obj = CS_NpcDoubleCultivateReq_pb2.CS_NpcDoubleCultivateReq()
45            message_obj.ParseFromString(decrypted_data)
46
47            result = {
48                "objId": message_obj.objId,
49                "type": message_obj.type,
50                "tenTimes": message_obj.tenTimes
51            }
52
53            print("解析结果:", result)
54            return result
55
56    except Exception as e:
57        print(f"Protobuf解析失败: {e}")
58        return None
59
60    def aes_decrypt_ecb(encrypted_data: bytes, key: bytes) -> bytes:
61        """AES-ECB解密"""
62        cipher = AES.new(key, AES.MODE_ECB)
63        decrypted = cipher.decrypt(encrypted_data)
64
65        # 去除PKCS7填充
66        padding_length = decrypted[-1]
67        return decrypted[:-padding_length]
68
69    def is_encrypted_protocol(protocol_id: int) -> bool:
70        """判断协议是否需要解密"""
71        # 这个逻辑需要根据游戏的EncryptFilter实现来确定
72        encrypted_protocols = {11125, 11126, 10003, 10005} # 示例
73        return protocol_id in encrypted_protocols
74
75    # 解析示例数据包
76    parse_game_packet("1400752b05e3ce1f940f618eb295ea6c6c9c26cc")

```

运行结果如下：

```

1   数据包长度: 20
2   协议ID: 11125 (0x2b75)
3   消息体长度: 16
4   协议名称: CS_NPC_DOUBLE_CULTIVATE_REQ
5   消息类型: CS_NpcDoubleCultivateReq
6   数据已解密
7   解析结果: {'objId': 237291675, 'type': 1, 'tenTimes': False}

```

看到这个结果，心里别提多高兴了。从APK分析到协议还原，整个流程走通了，数据包解析完全正确！

## 完整样本和代码参考

因为东西太杂和样本太大，全部上传到github，有兴趣的可以到github上查看

## 折腾完的一些感想

搞了一个多星期，总算把ZMxs这个游戏的协议给摸透了。回头想想这个过程，还是挺有意思的，踩了不少坑，但也学到了不少东西。

### 现在手游的技术架构



39

这次分析让我对现在手游的技术选择有了更清楚的认识。基本上主流游戏都是Unity做底子，然后业务逻辑全扔到Lua或者JS里去实现。这样搞确实有它的道理：



37

首先开发效率高得多，脚本语言写起来快，改起来也方便，不像C++那样编译半天。然后就是热更新这个杀手锏，服务端随时推个脚本更新，客户端马上就能用上新功能，根本不用重新发包。还有就是一套脚本能跑各个平台，省了不少适配的功夫。



41

但这样做也有代价。脚本这玩意儿相对来说比较好逆向，像这次我基本上把整个游戏逻辑都扒出来了。要是游戏公司把一些关键的数据计算或者反作弊逻辑放在客户端脚本里，那就给外挂开发者提供了很大便利。

### 关于资源保护这块

这个游戏的AssetBundle保护说实话挺一般的。就是简单的XOR异或，密钥还直接用文件名，这种保护强度对新手可能有点用，但对稍微有点经验的人来说基本没啥阻止作用。

要是我来做的话，至少得换成AES-256这种强一点的算法，密钥也不能这么简单粗暴。最好是搞个复杂点的密钥推导过程，再加上完整性校验，防止别人篡改资源文件。当然最根本的还是把重要资源放服务端，需要的时候动态下发，这样就算客户端被破解了也影响不大。

### 网络协议设计的一些细节

分析这个游戏的协议时发现了几个挺有意思的设计：

首先是加密策略不对称，客户端发送的数据可能会AES加密，但服务端返回的数据是明文。开始我还纳闷为啥这样设计，后来想想可能是性能考虑。服务端资源充足，解密不是问题，但客户端特别是低端设备，能省点计算就省点。

然后是选择性加密，不是所有协议都加密，而是通过一个EncryptFilter来判断哪些协议需要保护。这个设计挺实用的，敏感操作加密保护，普通操作明文传输，在安全性和性能之间找了个平衡点。

还有就是Protobuf + AES的组合使用。先序列化再加密，这个顺序是对的。Protobuf负责高效的数据序列化，AES负责数据保护，各司其职。

### 逆向分析的一些心得

这次分析下来，我觉得移动游戏的逆向大概有这么几个步骤：

先是引擎识别，这个很关键。看lib目录下的so文件基本就能判断出用的什么技术栈。确定了引擎就知道该用什么工具，走什么路线。

然后是代码层面的分析。Unity的话就用IL2CPPDumper，其他引擎有其他对应的工具。这一步主要是理解代码结构，找出关键的类和方法。

如果发现是脚本化架构，那重点就转到脚本提取上。Hook脚本加载函数是一种方法，但更彻底的还是找到脚本的存储位置，把完整的脚本库搞出来。

资源分析也很重要，特别是对于使用AssetBundle的游戏。搞清楚资源的加载流程，找到加密解密的关键点，这样就能把所有资源都搞到手。

协议逆向最好是基于源码分析，有了完整的脚本或者反编译代码，协议格式和加密机制基本上就一目了然了。

最后就是实战验证，拿真实的数据包来测试解析结果，确保分析的正确性。

整个过程下来，我觉得最关键的还是要结合静态分析和动态调试。光看代码不行，光Hook也不行，得两者结合才能突破各种保护。

### 一些想法

通过这次分析，我对Unity+Lua这种架构有了更深的理解。这种模式在手游行业确实很流行，开发效率高，热更新方便。但从安全角度来说，也确实存在一些问题。

不过话说回来，安全和效率本身就是矛盾的。游戏公司肯定是要在开发成本、运营成本和安全性之间找平衡。对于大部分休闲游戏来说，现在这种保护强度可能就够了。真正核心的数据和逻辑还是放在服务端比较安全。

随着技术的发展，保护和破解之间的对抗肯定还会继续下去。新的保护技术出来，新的破解方法也会跟上。这就是技术圈的魅力所在吧，永远有新的挑战等着你去解决。

完整代码查看: <https://github.com/wuhuaguo888/zmxs>

[培训]传播安全知识、拓宽行业人脉——看雪讲师团队等你加入！

☆

39

☆ 收藏 · 39 · 37 支持 分享

thumb-up

37

**赞赏记录**

gift

**参与人****雪币****留言****时间**

快乐的小跳蛙	你的分享对大家帮助很大，非常感谢！		0秒前
5m10v3	期待更多优质内容的分享，论坛有你更精彩！		26分钟前
岁月。	你的帖子非常有用，感谢分享！		1小时前
0x指纹	期待更多优质内容的分享，论坛有你更精彩！		1小时前
令狐双	感谢你的贡献，论坛因你而更加精彩！		2小时前
oMasko	感谢你分享这么好的资源！		2小时前
wx_Dispa1r	+5	你的分享对大家帮助很大，非常感谢！	2小时前
mb_kgfqphsx	非常支持你的观点！		2小时前
doduhuang	你的帖子非常有用，感谢分享！		2小时前
ldljlw	这个讨论对我很有帮助，谢谢！		2小时前
Mrack	为你点赞！		3小时前
便胜晴天	你的分享对大家帮助很大，非常感谢！		3小时前
mb_fruhjxwh	这个讨论对我很有帮助，谢谢！		3小时前
孤独的街	感谢你分享这么好的资源！		3小时前
ye1v	+1	为你点赞！	3小时前
dofla	你的分享对大家帮助很大，非常感谢！		3小时前
mb_rbbzqwvn	你的帖子非常有用，感谢分享！		3小时前
mb_dtidoewd	你的分享对大家帮助很大，非常感谢！		3小时前
清风jw	非常支持你的观点！		3小时前
ChuXin+.	非常支持你的观点！		3小时前

[查看更多](#)**最新回复 (7)****wuhuaguo888** 1

2楼

完整代码还在整理，实在太杂了，写的时候没有注意，晚点上传

15小时前

0 ...

## 最新回复 (7)



[mb\\_Idbucrik](#) 0

感谢分享

临时

3 楼

15小时前

0 ...



39



37



[MsScotch](#) 8

mark

极客

4 楼

14小时前

0 ...



[MaYil](#) 10

感谢分享

极客

5 楼

14小时前

0 ...



[Zepp7289](#) 0

感谢分享

临时

6 楼

3小时前

0 ...



[uni7corn](#) 7

感谢大佬分享

极客

7 楼

3小时前

0 ...



[mb\\_qimctavn](#) 9

6

极客

8 楼

3小时前

0 ...



快乐的小跳蛙

内容

高级回复

回帖 表情

返回

©2000-2025 看雪 | Based on Xiuno BBS

域名: 加速乐 | SSL证书: 亚洲诚信 | 安全网易易盾

看雪SRC | 看雪APP | 公众号: ikanxue | 关于我们 | 联系我们 | 企业服务

Processed: 0.040s, SQL: 51 / 沪ICP备2022023406号 / 沪公网安备 31011502006611号