2018年腾讯游戏安全技术竞赛进阶版writeup

转载

ruiruidad 于 2018-06-11 17:11:49 发布 974 收藏
文章标签： 腾讯 游戏技术 进阶版writeup

本来去年对比赛印象不错，所以今年还是在一堆事情的情况下挤了点时间强行通宵做了一下，题目本身还行，不过题目各种出错，主办方反馈极度慢也是无话可说。

最后本来是打算等比赛结果公布后抽空好好整理下把 writeup 发出来的，不过现在竟然由于提交答案时附件附错了，自己没注意检查，主办方也没有任何提醒，导致直接没有得分，实在是非常郁闷（真的是打过这么多比赛，输赢都无过），没有心情整理了，大家将就着看吧……

writeup 同时也放在 GitHub 上了，可以查看题目过程中的一些附件。

资格赛

不知道主办方是不是看我去年用 z3 angr 不太爽，或者看我复用 so 不太爽？上来规则里面就是不让有三方依赖，不让任何形式截取逆向汇编（IDA F5 结果可以复制么？？？）。
好吧，我们听主办方的，规规矩矩逆向，不偷懒跑符号执行了，一点代码也别复用了。
只是还真有点不清楚 Python 的库中哪些是自带的，哪些是 pip install 的了。。。。

按照流程来，首先 0x4864 对 key 做了检查，大致可以看出是要求是 39 位，其中 32 位是 hex digit，7 位是 `#`，并将 key 全部 `toupper` 后按照 `#` 将输入分段，大致可以感觉出是 4 个一组，一共 8 组。
接下来 0x496C 对这个 8 4 的输入，进行了惨无人道的简直毫无规律的各种小操作（+, -, `,`/,`,`%,`,`^,`,`&,`,`|）将输入转换成 5 个 qword，尤其配上这 cpp 的结构与冗余，整个代码不忍直视。
最好的方法无疑是符号执行或者直接复用代码，因为这个操作很显然在题目的目标下（根据 key 算 code），不需要进行逆运算。
但是为了满足出题人变态的要求，只能硬着头皮一点点看下来了，但不知道会不会有小的错误，毕竟没有办法进行完全的测试。
对于 standard 难度，0x7114 对输入 code 做了个变换，很明显可以看出是个换了表的 base64 decode（为什么都这么喜欢 base64，能不能来点新意？）。大概扫下后面的逻辑可知，这里最后得到的结果应该是 4 个 qword。

4. 0x5232 要求了上面一步最后一个 qword 是常量 0x32303138，接下来 0x5658 对剩余的 8(5 + 3) 个 qword 进行了校验。
   校验很明显是 3 个等式，显然这里用 solver 可以轻松愉快的解决，不过 solver 的库一般含有 binary，难以满足题目的使用要求，故而只好手解方程。
   大致可以转换为 `x ^ 2 == 0` 这种形式，于是按说可得唯一解。
   解方程过程中会有个除法，但对于本题，该除法可以整除，所以可以简单算得，不过暂不确定在模 2^64 的域下能否保证没有多解，数学都还给老师了，懒得算了，有一组解就好。

最后随机生成 key 测试了一会，没有发现 failed 的情况，就假装第二步细节没有逆错吧。。。

在 standard 的基础上，advance 只是在 standard 的 code 转换完后多调用了一个 0x6158，大致参数是 standard 转换完的 code 和常量字符串 `welcomegslab2018`。
首先会发现转换后的 code 长度应该是 16 的倍数，然后 16 个一组进行操作。
稍微调试一下，可以看出，常量字符串会扩展生成一张表，输入首先转换成 `4 * 4` 的矩阵，然后 xor 了表中数据，接下来进行了一个按行的 shift 操作，接下来就是一个 10 次的循环。
等等，看到这些操作，本能映入脑海的无疑就是 AES，带着这一猜想过一眼程序结构，可以很明显发现常量串正好 16 位，
最开始 expand key，然后 add round key，shift row，
接着循环 10 次，sub bytes，一堆奇怪 switch，add round key。
简直就是标准的不能在标准的 AES 加密流程了，那堆 switch 仔细看一眼常量或者干脆读一下每个 case，会发现用到的 9 11 13 14，正好就是 mix columns 时用到的多项式。
仔细检查一下各个常量表，sbox 256 个 byte 无重复，没问题，expand key 结果 176 个 byte，没问题，
不过 expand key 用的 sbox 貌似不是后面的 sbox，不过无所谓了，直接用 expand 后的结果就好了。
然后找个标准 AES 实现，换掉 sbox 和轮密钥，会发现结果不对。
仔细调试一下会发现，轮密钥的使用很奇怪，流程是加密的流程，但轮密钥是倒序使用（解密的方式），mix column 也是解密的参数，其次轮密钥的每四个 byte 中有两个反过来了。
改了半天 AES 各种对不上，感觉出题人把所有能换的表都给魔改了（给出题人深深地跪了），于是干脆从头跟着程序流程裸写一份好了，反正不复杂。
正向测试通过后，对着反着写一份，稍微麻烦点的就是 mix columns 需要将 14, 11, 13, 9 对应换成多项式的逆 3, 1, 1, 2。
最后在 standard 代码中算 code 时额外调用一下“解密”操作即可。

决赛第一题

运行程序可以发现就是个简陋的游戏地图，按照箭头返现移动一下会发现移不动了，并且看起来后面有东西，那么看代码吧。

首先 Manifest 可以看出是个 NativeActivity，打开 so 看下会发现，跟 Android 官网样例基本一样……
对着样例标好函数后，大概看一下 `engine_handle_input` 很明显限制了 `x <= 50.0`，直接 patch 掉移过去发现是个奇奇怪怪的形状。
然后左看看，右看看，也没感觉有啥特别的，就一堆 OpenGL 的操作，但感觉应该没啥牵扯到 flag 的。

无奈之下，只得深入看下 OpenGL 代码细节了，然而不得不说，这 OpenGL 的代码真恶心，感觉就是给我源代码，也看不懂……
去年从零开始学 mono，今年从零开始学 OpenGL，感觉再这样比几次，能把游戏技术全学一遍了……

找个 OpenGL 教程对着看，结果发现，texture 竟然跟教程上一样，em，这个题真是样例代码大杂烩……
按照执行顺序来，整个程序流程：

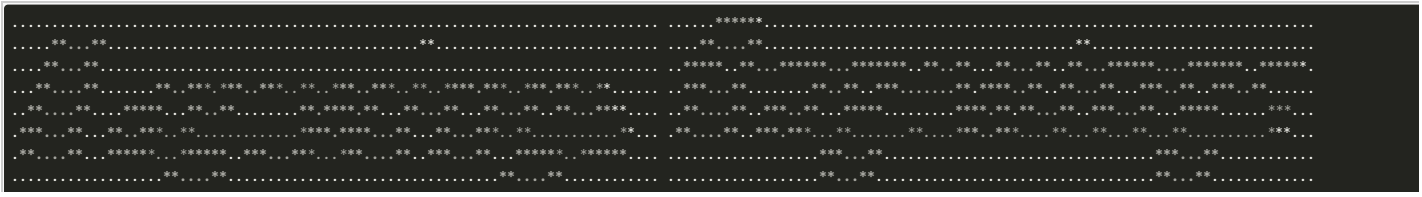
0x0002F420 是 init 函数，准备了顶点着色器和片段着色器的代码，然后初始化了一个 3057 个常量 float 的数组；
`android_main` 里面设置了 `engine_handle_cmd` 和 `engine_handle_input`，然后消息循环处理，无操作时直接调用 `draw (0x00030094)` 更新界面；
`engine_handle_input` 大致处理了一下滑动的操作，更新全局的当前坐标，同时会限制可移动的坐标范围；
`engine_handle_cmd` 大致处理了一下界面初始化的操作，加载了 `container.jpg` 和 `awesomeface.png` 作为着色器中的 texture，从 0xA18D8 加载了 180 个常量 float 做为顶点信息；

5. `draw` 中设定了 projection、view、model 三个变换矩阵并绘制界面，其中 view 由当前全局坐标算出，大致反应当前视角，model 由之前 3057 个 float 的常量数组决定，3 个一组，生成一个位移的矩标），同时结合上一个与分组 index 相关的旋转矩阵（达到类似随机旋转的效果），一共 1019 个 model。每个 model 会根据之前的 180 个顶点信息对应绘制 12 个三角形（每个顶点含有 3 维坐标及`180`），应该是对应画了一个正方体的 6 个面。通过调试改变 model 个数，可以看出前一半画出了之前右上角的不规则图形，后一半画出了中间的箭头。

提取下这 1057 个 model 的坐标。很明显可以看出坐标分成了两个部分，前 361 个是绘制那片奇怪区域的，剩下的是绘制箭头的。

对着坐标看了一会，没发现什么异常，于是仔细研究坐标含义。
首先坐标根据之前看程序的感觉，结合动态调试（拿着一个点控制变量法调节三维坐标的值）观察，可以发现，
这是一个球面坐标（不妨当做地球好了），三维分别是经度、纬度和到球面的距离（-90 大概是球心，故第三维限制为最小 -89）。

既然是球面坐标，继续对着坐标脑洞，考虑按 z 坐标分层，于是写了成脚本，按 z 坐标范围过滤，二维打点画图，试了半天后发现，貌似完全不分层的时候结果非常合理，咋看咋像 flag：



```
.....******.....******.....
```

瞬间感觉自己智障了，一直以为题目是要在不调试、不篡改程序的情况下，通过一些特定的移动方式，看到 flag，
其实，题目是想让我们换个视角来看地图，而程序本身大概是不能换到这个视角的。

虽然看到了 flag，但感觉这自己二维做图估计不会过，那考虑怎么在程序中转换视角。
继续调试，仔细观察下移动后更新的三维坐标和各个变换矩阵发现，我们的滑动约等于上下左右平移这个球，
view 矩阵是做了这个平移的操作，project 矩阵才是真正的视角矩阵，相当于放在球体外全局 z 轴上一固定地方相机，默认视角 45 度。

为了看到 flag，我们首先将全局记录的三维坐标改成 `(0, 0, -1)`，此时得到 view 矩阵：

```
1 0 0 0 0 1 0 0 0 0 1 -3 0 0 0 1
```

对于球面坐标而言，旋转比平移更容易，我们计算下箭头和 flag 坐标的平均值的差，
大致可以得出绕全局 y 轴旋转 -59°，绕全局 x 轴旋转 -17.5°，为了防止图案过大，我们可将球沿 z 轴负方向平移 -90，
于是最终得到 view 矩阵：

```
1 0 0 -59 0 1 0 -17.5 0 0 1 -90 0 0 0 1
```

修改好后，让程序完成绘制，可以看到 flag，见图。

决赛第二题

Level 1

首先看下 Manifest，查一查会发现程序是一个 UE4 引擎做的游戏，由于上了个大型引擎，相比于之前的题程序大了很多，手头的破测试机跑起来卡的飞起……

大概查了下资料，想找到用户代码入口，半天无果，全局搜了下字符串，也找不到游戏中的字符串，只能抱着最后的侥幸乱翻下 so 了。
首先搜了下名字，感觉 `libph2.so` 和 `libtmgs.so` 比较可疑，打开翻下 export，发现 `libtmgs.so` 中有 `maln`、`check` 之类的函数名，瞬间对出题人充满了感激。
果断下断跑起来，毫不意外的点击验证后成功断下来，终于可以开始分析了。

调了下很明显发现一个 luac，`maln` 通过调用 luac 的 `check` 函数来校验输入。

dump 下来，查了下 luac 相关信息，发现貌似和流传的版本号对不上？！格式不对劲解不出来啊！

乱翻看到 lua load 相关函数，于是对照着 lua 5.3 源码打了一圈符号，蛋疼的发现，程序和源码的文件格式基本一致，除了开头的版本号被从 0x53 改成了 0x11……
给网上搜出来的破教程跪了，教程中的打着 5.3 的招牌，讲的绝对不是 5.3 的格式，坑啊坑啊！

可是既然 load 没有问题，那么改掉版本号后用搜到的针对 5.3 的 luadec 尝试，程序还是直接崩，无奈 GDB 一下 luadec，发现是指令翻译的时候不对……
瞬间感觉又有去年的既视感，只是把 `.NET` 换成了 `lua`，于是继续对着源代码找执行部分代码 `luaV_execute (0x18b60)`。
稍微对比下发现，操作数的切割应该没有动过，貌似和去年一样把指令顺序动了……
再仔细看一下发现，比去年更坑的是，今年貌似不像去年直接循环移动了下，case 63 个，其中有重复，去重后剩 47 个，而正好源码中 opcode 有 47 个。

感觉要一个个把指令对上来实在要命，突然想起来，还有个基础版，可以参考一下，大概估计有哪些指令。
瞬间感觉偷鸡不成蚀把米，本来看时间不够了，直接看的进阶版，结果又跟去年一样饶了一圈弯路（为什么年年都是只篡改 Opcode，不篡改 loader！！！）。

打开基础版，果然 Opcode 没有动，连前面那个忽悠人的版本号都没有，直接标的 5.3 版，
血崩，感觉自己真是傻啊，有 easy mode 不走非要走 hard mode，下次一定好好参考基础版做题！

用 luadec 跑下基础版，发现只能 disassemble 不能 decompile，调了下大概修掉最外层 `upvalue->name`（大概是我们的输入？）不存在的问题，还是有问题，于是决定放弃。
程序函数有点多，不过既然之前觉得调用的 check 函数，那么直接收下 check，发现是个构造的闭包，貌似很符合设定。
随便看两眼，就会发现估计是用 `F8998657AFE06DD5AA593D88FB3DB3E4` 作为 key，加密结果一位位校验，大概是 `\x1e\xc9\x86\x8b3h\xd1\xa4\xad{\x86t\x07\x1c\xeen\x87x\x81Gk\xbb\xed\x98o\xca\xda\xc0\xd`
网上搜个 rc4 跑一下，果然解密成功得 `C3F6B4473DB70B38B554F6F3C2E6058C`，输入程序校验无误。

再回来看进阶版，直接看 luac 会发现同样有着像 key 的串，以及 32 个用来对比的常量整数，如下：

```
04 21 43 44 44 38 41 41 41 41 35 30 30 43 41 38 45 46 38 37 31 33 45 31 43 37 35 38 31 37 35 30 30 33 13 C4 00 00 00 00 00 00 00 13 F3 00 00 00 00 00 00 00 13 E4 00 00 00 00 00
00 00 00 13 C6 00 00 00 00 00 00 00 13 9D 00 00 00 00 00 00 00 13 5E 00 00 00 00 00 00 00 13 12 00 00 00 00 00 00 00 13 45 00 00 00 00 00 00 00 13 1B 00 00 00 00 00 00 00 13 34
5B 00 00 00 00 00 00 00 13 44 00 00 00 00 00 00 00 13 A2 00 00 00 00 00 00 00 13 CD 00 00 00 00 00 00 00 13 9B 00 00 00 00 00 00 00 13 38 00 00 00 00 00 00 00 13 F1 00 00 00 00
00 00 00 00 13 74 00 00 00 00 00 00 00 13 9E 00 00 00 00 00 00 00 13 4D 00 00 00 00 00 00 00 13 6F 00 00 00 00 00 00 00 13 42 00 00 00 00 00 00 00 13 98 00 00 00 00 00 00 00 13
13 AE 00 00 00 00 00 00 00 13 54 00 00 00 00 00 00 00 13 7B 00 00 00 00 00 00 00 13 EA 00 00 00 00 00 00 00 13 85 00 00 00 00 00 00 00
```

即 key 是 `CDD8AAAA500CA8EF8713E1C758175003`，加密结果是 `\xc4\xf3\xe4n\xc6\x9d^\x12E\x1b4[D\xa2\xcd\x9b8\xf1"t\x9eMoB\x98g\xaeT{\xea\x85`，
然后惊奇的发现解密结果不对，看来出题人防猜了？

仔细一看，常量整数不满 32 个，看来常量有重复，那扣指令表：

```
71 00 00 00 1A 40 40 80 42 C0 40 00 82 00 40 00 7D 80 00 01 1A 40 00 81 1A 00 00 82 42 40 41 00 87 00 00 00 7D 80 00 01 1A 40 00 82 42 80 40 00 82 00 41 00 79 80 00 01 1A 40 00
00 E9 40 02 00 3C 81 02 00 68 C1 02 00 A8 01 03 00 E2 41 03 00 28 82 03 00 68 C2 03 00 A9 02 04 00 E8 42 04 00 28 83 04 00 62 C3 04 00 A2 03 05 00 E9 43 05 00 29 84 05 00 7C C4
06 00 22 85 06 00 62 C5 06 00 A8 05 07 00 E8 45 07 00 3C 86 07 00 69 C6 07 00 BC 06 08 00 E9 46 08 00 3C 87 08 00 68 C7 08 00 BC 07 09 00 E8 47 09 00 29 C8 04 00 69 88 09 00 4C
00 CA 93 1A 80 CA 94 42 40 4A 00 82 C0 4A 00 8D 00 4B 01 C2 80 41 00 B7 80 00 01 35 80 80 00 09 C0 04 80 42 C0 4A 00 4D 80 CB 00 82 80 41 00 C2 40 4A 00 02 41 4A 00 79 80 00 02
82 C0 4A 00 8D C0 4B 01 C2 40 4B 00 A3 80 00 01 E9 00 0C 00 60 C0 80 00 1A 40 80 93 42 40 4A 00 71 80 CA 00 1A 40 80 94 50 24 03 22 05 00 F9 7F 1A 80 CA 94 42 40 4A 00 82 C0 4A
00 B9 80 00 01 35 80 80 00 09 80 05 80 42 C0 4A 00 4D 80 CB 00 82 80 41 00 C2 40 4A 00 02 41 4A 00 77 80 00 02 1A 40 80 96 50 1B 03 42 42 C0 4A 00 4D C0 CB 00 82 40 4B 00 77 80
4A 00 8D C0 00 01 70 80 80 00 25 40 00 80 7B 00 00 00 44 00 00 01 42 40 4A 00 71 80 CA 00 1A 40 80 94 09 40 F8 7F 90 39 83 38 7B 00 80 00 44 00 00 01 04 00 80 00
```

提取下 Opcode：

```
26 2 2 61 26 26 2 7 61 26 2 2 57 26 31 60 41 60 40 40 34 40 40 41 40 40 34 34 41 41 60 34 60 34 34 40 40 60 41 60 41 60 40 60 40 41 41 12 26 26 26 2 2 13 2 55 53 9 2 13 2 2 2 57
49 26 16 5 26 2 2 13 2 57 53 9 2 13 2 2 2 55 26 16 2 13 2 55 2 2 13 48 37 59 4 2 49 26 9 16 59 4 4
```

而基础版 Opcode 是：

```
8, 6, 6, 36, 8, 6, 6, 36, 8, 8, 6, 6, 36, 8, 11, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 43, 8, 8, 8, 6, 6, 7, 6, 36, 33,
6, 7, 6, 36, 6, 6, 7, 31, 30, 3, 38, 6, 13, 8, 30, 3, 38, 38
```

很显然，没有中间那一串相同的 Opcode，即 `OP_LOADK`。
看来还是得研究下 Opcode 的变换了，不过有了基础版本的代码参照，寻找对应关系就变得异常容易，基本看一下常量、结构等特征就可以确定 Opcode 了。

==，突然想起来，在进阶版里，部分 Opcode 有重复，立刻找到 `OP_LOADK` 对应的地方，果然有重复，`34 40 41 60` 都是，替换一下得：

```
26 2 2 61 26 26 2 7 61 26 2 2 57 26 31 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 12 26 26 26 2 2 13 2 55 53 9 2 13 2 2 2 57
49 26 16 5 26 2 2 13 2 57 53 9 2 13 2 2 2 55 26 16 2 13 2 55 2 2 13 48 37 59 4 2 49 26 9 16 59 4 4
```

果然中间出现了我们要的特征，正好有 32 个，提取操作数得 `8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 19 38`，
那么加密结果应该是 `\xc4\xf3\xe4n\xc6\x9d^\x12E\x1b4[D\xa2\xcd\x9b8\xf1"t\x9eMoB\x98g\xaeT{\xea[\x85`。

然而直接解密结果还是不对，再仔细看下会发现 check 的常量中多了一个 `prexor`，看来预处理了下输入。

仔细对比发现另一个串 `E0EA72E0E1C1BFFBC26E8B47AD9D809C`，应该是跟 `prexor` 相关，但直接 xor 到结果上还是不对。
以及顺便看到 `Lua 5.2` 的字符串，瞬间感觉心凉了，可我记得试过 5.2 死的更惨啊……

感觉自己不适合猜谜，于是还是规规矩矩把 Opcode 对了下，得到对照表 `{2: 6, 3: 42, 4: 38, 5: 30, 6: 0, 7: 0, 8: 5, 9: 30, 10: 30, 11: 47, 12: 43, 13: 7, 14: 7, 16: 47, 17: 5, 18: 4, 20: 45,
8, 27: 0, 29: 10, 30: 15, 31: 11, 32: 29, 33: 41, 34: 1, 35: 36, 36: 19, 37: 30, 38: 32, 39: 37, 40: 1, 41: 1, 42: 35, 44: 39, 46: 44, 47: 14, 48: 31, 49: 13, 50: 40, 51: 26, 52
56: 34, 57: 36, 59: 3, 60: 1, 61: 36}`

decode 后发现，prexor 原来是异或了一个新的常量串 `\xb7s\x80d\n\xccQ\x0f\x1dXCw,h\xca\x07\x9c\xa4Uw\x03\x9d\xd0\xbf\xfde\x02\xac\xf8\x83i2`，
尝试解密发现还是不对，看了半天不知道为啥，但发现之前那个 `E0EA72E0E1C1BFFBC26E8B47AD9D809C` 串貌似没用，就莫名其妙刚开始但参数调用了一下 check。
于是无聊的试了下当输入，em，竟然过了，wtf? 这是题目又意外了，测试代码没删么……
既然答案都出来了，也懒得 debug 了，真是给出题人深深的跪了，敢不敢出题的时候认真点？

Level 3

第一关通过看字符串猜出了检查函数，第二关就没有这么幸运了。
于是参考第一关的调用栈，在调用栈上一层层向上下断，在第三次 ret 之后的调用点断住了。
不过断住次数太多，大概是什么消息处理函数之类的，于是找一些不触发第二关的 check 逻辑的操作记录一些调用黑名单，
在去除掉会被多次调用到的函数，就剩了大概四个函数，人工试试筛选一下，貌似看着都不太对劲……
突然想起来，其实最好的办法应该是直接 trace 或者内存断点，看什么时候跳出了 `UE4.so` 的范围，十有八九就是目标……
结果 IDA 不能 Android 上貌似下不了内存断，要去下个 gdb，然后 IDA 的指令级 trace 貌似也不 work，感觉真是给 IDA 跪了，要用的时候全 GG。

无聊的看了下第一个返回的函数，发现其中很明显出现了一个 `pfn_check_key2` 的符号，之前第一关调用的就是这个东西。
em...感觉自己是智障，竟然没查 UE4 的符号，还以为这个 so 所有 UE4 程序都是一样的。
于是接下来发现了 `pfn_first_round` 和 `pfn_check_key1` 两个符号，纳尼，我刚才做的真的是第一关么，这会各种慌！

了是接下来发现了 `pfn_first_round` 和 `pfn_check_key4` 两个符号，纳尼，我刚才做的真的是第一关么，这都看啥情况？

调一下，发现 `pfn_first_round` 和 `pfn_check_key4` 分别指向 `libtmgs.so` 中的 `null` 和 `ths` 两个函数，
见了鬼了，这两个函数断过，断不下来的啊？！
难道说，第一关真的应该是 `null`，然后我真的直接做了第二关，结果程序就 bug 了？？？
可这第一关我咋还能输入第二关的结果过的……虽然好像确实不太对劲的是秘钥最后可以删除几位仍旧能过……

不知道发生了什么实在触发不了断点，但是 `pfn_check_key4` 还没有被用过，那么一定会被用，
刚才人工过滤的列表中，有个 `sub_62BEBDA0` 中有句 `*v4 = sub_627BE820((int)&v11, (int)&v8);` 比较可疑，改 `*v4` 里的值为 1 是可以直接过掉第二关的。
那么先跳了试下第三关，果然 `pfn_check_key4` 在第三关被触发了……

第三关 `ths` 首先把 `libph2.so` 载进来调用 `punkHash_check`。
em…这个 `libph2.so` 还真是不能更友善了啊，直接把 lua 引擎的 symbol 都给了，luac 也是直接常量给出。
（以及，这个 `ths` 很神奇的地方在于，如果 `punkHash_check` 找不到的话，会直接调用之前 `maln` 里面调用的那个 check 函数，不知道什么用意……）

luac 拉出来一看，咦，这个头完全不对啊，以及这个 load 函数点进去看两眼完全不对劲啊，查了一下发现，这个原来是 luajit 的头。
搞来搞去，最后还是用 ljd 直接搞出一个反编译的代码，不过效果真的很差，也就勉强能看一下。

蛋疼的开始几千行 lua 逆向之旅，不过好在大部分看起来都应该是库函数，有名字，直接折叠掉不看，然后从 check 函数看起。
最后发现其实关键的就是一堆名字跟被混淆过似的函数，其实这些名字基本就是做了裸的字符串替换混淆，各种相同前缀，不过这都无所谓了。
稍微认真点看一下结构，会发现这就是实现了一基于栈的虚拟机，反编译结果中有个 opf 数组存放了 Opcode 与处理函数的对应关系，操作都很简单，很好逆。
而 check 的功能就是对输入，跑固定的虚拟机代码，输出和常量
串 `ETKdgxteV6FHLzDCwmaVb9pYU5kSV6paNicOnO/wA0ZzM4CzVmALImn0CmxRhx0xSq/jV3Ad9i6s4+jQF0TUY3vCVm2obdcm8Oozofmj1nCCVPBoT7qk+2n+bzN+jhz6VPJEw8OkfkuCoGRJRlftVYv+6uwKRYPza/RnlFVfVkgw+zo`

那首先写个反汇编器把虚拟机代码反汇编一下就好了，不过也不知道自己咋就脑抽的写了个 emulator，算了，也无所谓了，把执行 trace 打出来还是勉强能看的。
看的时候发现代码有点诡异，可能是部分控制流指令翻译的有点问题。
这时候想到这个 luajit 既然没有篡改过啥，那直接就可以 load 起来跑，现学了一下 lua 语法，写了个测试代码果然跑起来了。
然而默认是没有输出的，想修改代码估计也麻烦，但其实，在程序中埋了几个输出点，只是输出函数 LOGPH 没有实现功能。
于是强行换掉 LOGPH 为 print，再跑就能看到程序中间的日志，非常愉快。

这时候对比一下输出，会发现几个中间打印量 emulator 跑的结果都是对的，不过就是输出不太正常，
但大概根据逆向的情况估计一下逻辑，输出应该就是把中间打印的这些量转成的字符串，转换方式无非就是看做 64 进制的数后用虚拟机中常量表 `VChf+BoN8qw43JzinLRQm95F/u7D6M0bYIeSTypAktsjOgWE2dUHrlGaPK1cZXvx` i
根据猜想，验证一下无误，剩下的工作就是进行逆操作了。

首先 8 个 byte 一组，假定前 2 个构成 x，后 6 个构成 y。那么在只考虑可见字符串输入的情况下，四个数字可以分别表示如下：

```
# input: ABCDEFGH x = 0x4241 # AB y = 0x484746454443 # CDEFGH k1 = 8 * x**3 + 13 * x**2 + 26 * x + 87 k2 = y % 61454 * 256 k3 = y % 54732 + y % 5136 % 256 * 256 * 256 k4 = y %
256
```

其中 k1, k2, k3, k4 分别转换成 8, 4, 4, 4 个 byte 到输出。

算 x 本来需要解三次方程，但鉴于数据范围小，可以直接穷举（提前把表打好可以免得算太慢）。
算 y 的话，首先根据 k2 可知 y 除 61454 的余数，
k4 中 `y % 5136 / 256` 最大为 20，`y % 25548 * 256` 是 256 倍数，故可以拆解，得知 y 除 25548 的余数，及 `y % 5136` 除 256 的商，
k3 中 `y % 54732` 最大为 54731，`y % 5136 % 256 * 256 * 256` 是 65536 的倍数，故也可以拆解，得 y 除 54732 及除 `y % 5136` 除 256 的余数。
故 `y % 5136` 可以确定于是对 61454，54732，25548，5136 用中国剩余定理即可求解 y。

最后解得字符串：`The MIT License (MIT)\tCopyright (c) 2015  <gslab@tencent.com>..\tPermission is hereby granted, free of charge, to any person o  youy\tof this software and ass`
`(key:8638599518A635CCC0734ABF55038747)hout restriction, including without limitation the rights\tto use, copy, modify, merge, publish, distribute, sublicense, and/or sell\tcopies`
`permit persons to whom the Software is\tfurnished to do so, subject to the following conditions:`，输入即可过第三关。

Level 2

首先仔细检查下之前怀疑的函数会发现，之前说的 `sub_62BEBDA0` 中有的 `*v4 = sub_627BE820((int)&v11, (int)&v8);` 其实是做了个 `strcmp`，但是两个待比较函数来历不明。

正常逻辑下内存断往回跟应该挺容易的，不过这里是 Android，下断很受限制（看来应该做 Windows 版？可我没有 Windows 啊！！！）。
Android 下不能硬断，IDA 指令 trace 不正常，也不能 watch，传了个 gdb 上去后软 watch 多线程下各种不太正常，`set scheduler-locking on` 锁住线程切换后会稍微正常一点，勉强可用。

总之最后就是一通 watch 死命往回追踪数据来源，终于发现原来是 `sub_616E0F40`，这就是之前那个点第二关额外多出来的函数，但是调用了两次，当时犹豫了一下没考虑。
这也很符合上面说的，是算出了两个字符串对比。
稍微乱点一下，会发现里面很显然有个 md5，然后算下 `md5(input)` 果然等于待比较的第一个串。
咦，之前乱折腾的时候，发现出现过程序中有函数是 md5，还专门算过试了下，记得当时试下来不是第一个串啊，gg。

直接把第二个串扔给逆 md5 的网站果然失败了，但仔细一想，既然 md5 不可逆，程序用 md5 的结果做对比，那其实含义就是要求和算 md5 之前一直，
而待比较的 md5 既然是算出来的，那么我们断在算 md5 处看下输入就好。

算 md5 函数大概还是 `sub_616DFB58`，断下来，大概看一下，第二个参数是指向待算字符串的二级指针，于是 dump 下来（貌似点一次断了 3 次，分别是第一关 key，第二关 key，输入）。

```
0x6b28a280: 0x00000074  0x00000065  0x0000006e  0x00000063 0x6b28a290: 0x00000065  0x0000006e  0x00000074  0x0000005f 0x6b28a2a0: 0x0000006d  0x0000006f  0x00000062  0x00000069
0x0000006c  0x00000065  0x0000005f  0x00000067 0x6b28a2c0: 0x00000061  0x0000006d  0x00000065  0x0000002b 0x6b28a2d0: 0x0000002d  0x00000039  0x00000039  0x00000039 0x6b28a2e0:
0x00000038  0x00000039  0x00000033  0x00000038 0x6b28a2f0: 0x00000038  0x00000037  0x00000000  0x00000011
```

即输入是 `tencent_mobile_game+-999893887`，输入验证无误。这么看来这关竟然没出 `libUE4.so`，还真是万万没料到。不过这下算是把 GDB 断点功能好好又熟悉了一遍，附个最后残留的断点列表。

```
(gdb) i br Num     Type           Disp Enb Address    What 1       breakpoint     keep y   0x62bebe5c     breakpoint already hit 32 times 10      breakpoint     keep y   0x62be
only in thread 15     breakpoint already hit 27 times 17      watchpoint     keep n              *(int *)0x64cc8e18     breakpoint already hit 2 times 18      breakpoint     ke
15     stop only if $r0 == 0x64cc8e18 (target evals)     stop only in thread 15     breakpoint already hit 38 times 23      watchpoint     keep n              *(int *)0x69a5682
in thread 15     breakpoint already hit 1 time 30      watchpoint     keep n              *(int *)0x6b3eb6e4 thread 15     stop only in thread 15     breakpoint already hit 2 t
   keep n   0x618ae274     stop only if ($r3 & 0xfff) == 0xd1c (target evals)     breakpoint already hit 27 times 39      breakpoint     keep y   0x618ae274     stop only if ($
(target evals)     breakpoint already hit 17 times     ignore next 98 hits 42      breakpoint     keep y   0x616dfb58     breakpoint already hit 12 times
```